HARDWARE ACCELERATION OF THE SAMTOOLS VARIANT CALLER

BY

BRETT WALKER DUTRO

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Adviser:

Professor Steven Lumetta

ABSTRACT

This thesis presents a design for a hardware-accelerated implementation of the SAMtools variant caller on an FPGA. It also includes a performance analysis of the algorithm and a proposed change to its software architecture to improve performance.

SAMtools is normally invoked as a two-step command, where the results of `samtools mpileup` are piped into `bcftools call`. Profiling their execution revealed that the single most computationally intensive part of the algorithm is the function `bcf_call_combine`, which is responsible for 23.63% of the execution time of `samtools mpileup`. In addition, the various functions used for output in `samtools mpileup` are responsible for a total of 24.58% of its execution time, while functions responsible for handling input in `bcftools call` accounted for 93.34% of the execution time of that program. Profiling the full command revealed that `bcftools call` was only responsible for 5.82% of total run time.

Both software and hardware approaches were taken to improve the performance of SAMtools. The software approach combined the two parts of the variant calling command into a single executable called `mpileup_call`. The hardware approach implemented the functions `bcf_call_glfgen` and `bcf_call_combine` in a Verilog/SystemVerilog design targeting an Altera Stratix V GX A7 FPGA.

Combining the two parts of the command into a single program resulted in a 2.42x speedup. The hardware accelerated version of the combined tool achieved an overall speedup of 2.93x over the base SAMtools workflow.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The advent of next-generation sequencing (NGS) has drastically reduced the time and cost needed to sequence an individual's genome [1], [2]. As costs drop, the prospect of personalized medicine based on genomic data is becoming more and more exciting [3]. One key aspect of personalized medicine is the need for rapid genome sequencing and analysis. High speed sequencing has been achieved, with one group having recently performed alignment and variant calling for 240 genomes in 50 hours [4]. It should be noted, however, that this result required the use of a Cray XE6 supercomputer. For personalized medicine to become a reality, doctors and hospitals will need access to this level of speed without having to invest in a supercomputer.

Hardware accelerators provide an answer to the problem of lowering hardware costs while increasing the speed of genomic data analysis. Prior attempts to accelerate genomic workloads have been mostly GPU-based. GPUs are excellent for accelerating parallel floating point workloads due to their many on-chip floating point units. In addition, they allow for high flexibility and rapid development because they are fully configured in software. However, GPU acceleration also has disadvantages, namely high power requirements and poor performance in the presence of branch divergence.

The other main approach to hardware acceleration is the use of field-programmable gate arrays (FPGAs). FPGAs are reconfigurable hardware devices that are configured with logic gate-level designs written in a hardware description language such as Verilog or VHDL. FPGA-based accelerators can provide high speedups for parallel workloads with much lower energy costs than GPUs, but require much more time to develop. They also have historically been poor choices for heavy floating point workloads due to the

high area costs of floating point arithmetic units, but FPGA manufacturers have begun adding specialized hardware [5], [6] to implement these functions efficiently.

The main contributions of this work are:

- An in-depth analysis of the performance of the SAMtools variant calling workflow.
- A proposed change to the software architecture of SAMtools to improve variant calling performance.
- A design for an FPGA-accelerated implementation of SAMtools written in Verilog and SystemVerilog.

This thesis is organized as follows. Chapter 2 provides background on next-generation sequencing and variant calling, along with a summary of related work. Chapter 3 gives an analysis of the performance of the SAMtools variant calling workflow and a summary of its major algorithmic steps. Chapter 4 presents the software improvements made to SAMtools along with the design of the hardware accelerator. Chapter 5 presents the performance improvements from the architectural modifications to the software and the hardware accelerator. Chapter 6 discusses the results in the context of other hardware accelerators targeted at computational genomics and suggests future work in this area.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Next-Generation Sequencing

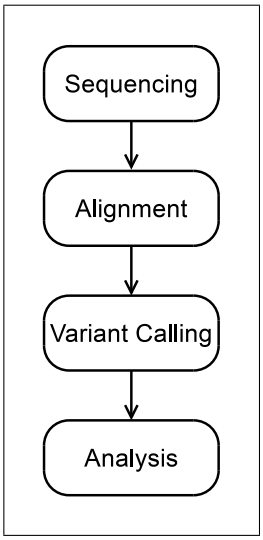

Figure 2.1: Typical variant calling workflow.

Next-generation sequencing methods provide fast, low-cost genotyping of individuals. In broad terms, a sequencing workflow consists of four major steps, illustrated in Figure 2.1. *Sequencing* is the process of acquiring genomic data from biological samples. Deoxyribonucleic acid (DNA) stores genetic data as a sequence built from an alphabet of four nucleotides: adenine (A), cytosine (C), guanine (G), and thymine (T). The nucleotides form complementary pairs: A with T and C with G. High throughput DNA sequencers randomly cut many copies[1] of DNA molecules into fragments called short reads. The complements of these reads are then rebuilt with specially

[1]The number of copies is referred to as the *coverage* or *depth* of sequencing.

treated fluorescent nucleotides. By reading the fluorescence intensity of these complementary reads, sequencers can determine the bases making up the original strand of DNA [7], a procedure known as base calling. Associated with each of these read bases is a *quality score* that represents the confidence of the base call.

Due to the random nature of the cuts performed in sequencing, the position of the reads within the individual's genome is unknown. Therefore, it is necessary to perform the second step in the workflow, *alignment*. Alignment is the process of mapping short reads to a reference genome for the species of the individual being sequenced. Some variation of the Smith-Waterman algorithm [8] is typically used for alignment, in many cases augmented with the Burrows-Wheeler Transform [9]. Figure 2.2 gives an example of the results of alignment. After alignment, variant calling (discussed in depth in

```
REF: ATTCAATAAACGGTGCTTGGGTAGCTGGCTAACCATATGCAGAAGAATGAAAGT

001: ATTCAATAAACGGTGCTTGGGTAGCTGGCTAACCATATGCAGAAGAATGAAAGT
002:                  TTGGGTAGCTGGCTAACCATATGCAGAAGAATGAAAGT
003: ATTCAATA                      CTAACCATACGCAGAAGAATGAAACT
004: ATTCAATAAACGGT
005: ATTCAATAAACGGTGCTTGG
006: ATTCAATAAACGGTGCTTGGGTAG
007: ATTCAATAAACGGTGCTTGGGTAGCTGGCTAACCATATG
008: ACTCAATAAACGGTGCTTGGGTAGCTGGCTAACCATATGCAGAAGAATGAAAGT
009: ATTCAATAAACGGTGCTTGGGTAGCTGGCTAACCATATGCAGAAGAATGAAAGT
010: ATTCAATAAACGGTGCTTGGGTAGCTGGCTAACCATATGCAGAAGAATGAAAGT
011: ATTCAATAAACGGTGCTTGGGTAGCTGGCTAACCATATGCAGAAGAAATAAAGT
012: ATTCAATAAACGGTGCTTGGGTAGCTGGCTAACCATATGCAGAAGAATGAAAGT
013: ATTCAATAAACGGTGCTTGGGTAGCTGGCTAACCATATGCAGAAGAATGAAAGT
014: ATTCAATAAACGGTGCTTGGGTAGCTGGCTAACCATATGCAGAAGAATGAAAGT
```

Figure 2.2: Example of alignment results. REF is the reference genome and 001-014 are the aligned reads.

the next section) determines the most likely base when there are mismatches between the reads and the reference. The final step, analysis, generally requires human interaction. Human analysis of variant calling results allows

for filtering of false positives and provides deeper insight into how genetic variation is reflected in phenotypes.[2]

2.2 Variant Calling

The expected error rate of NGS sequencing can vary from 0.26% to 12.86% per base call depending on the technology used [10]. One study [11] also found that the read error rate for Illumina Hi-Seq machines can reach 8.83% at certain positions in the PhiX genome, while the expected error rate according to specifications is only 0.26% [10]. Alignment can also introduce errors; one study [12] across several species found a misalignment rate ranging from 0.33% to 1.1%. One method of compensating for these errors is increasing the coverage to upwards of 20x [7], but this increases both the cost and time of sequencing. Variant calling algorithms, on the other hand, provide a computational method to compensate for the aforementioned errors in NGS data. Most modern variant callers (including SAMtools) use a probabilistic framework that incorporates Bayesian inference [7].

2.2.1 Bayesian Inference in Variant Calling

Bayesian inference utilizes Bayes' theorem (2.1),

$$P(X|Y) = \frac{P(Y|X)P(X)}{P(Y)} \tag{2.1}$$

where $P(X|Y)$ is the posterior probability, $P(Y|X)$ is the likelihood, and $P(X)$ is the prior probability. In the context of variant calling, X is the actual base at the current position of the sequenced genome and Y is the set of observed bases in the aligned reads. $P(Y)$ is assumed to be independent of the true base.[3] Therefore, it is treated as a constant scaling factor. The

[2] A phenotype is the set of observable physical characteristics of an organism.
[3] $P(Y) = \sum_i P(X_i)P(Y|X_i)$.

most likely base $\hat{X}$ of an alignment can therefore be found with (2.2).

$$\hat{X} = \arg\max_X \frac{P(Y|X)P(X)}{P(Y)} = \arg\max_X P(Y|X)P(X) \tag{2.2}$$

These probabilities at each position are assumed to be independent [13]. The human genome is approximately 3 billion base pairs long, and the calculations involved in variant calling must be performed at every position along the genome. This calculation results in a large computational workload that can be easily parallelized.

2.2.2 Types of Variants

There are several different classes of variants [14]. The simplest type is the single nucleotide polymorphism (SNP). SNPs occur when a single nucleotide differs with respect to the reference genome (e.g., the reference genome has an A at a certain position, but a sample has a T at that position). The second type is an insertion or a deletion (indel). An indel could be an insertion of a short string of nucleotides with respect to the reference genome, or a sample that is missing a short string that is present in the reference (e.g., the reference is AACGTGA and a sample is ATGA). These two types of variants are easier to detect than structural variations (SVs), which are large-scale insertions or deletions, and copy number variations (CNVs), where a certain string is repeated some number of times.

2.3 Related Work

The advent of NGS workflows is still relatively recent, and as a result there have not been many attempts at hardware acceleration. Most of these efforts have focused on GPU-based accelerators for alignment and assembly, as they constitute the most time-intensive part of the workflow. Some (but not all)

of these GPU aligners include: MUMmerGPU [15], BarraCUDA [16], and SOAP3 [17]. MUMmerGPU is a CUDA[4] implementation of the Mummer [18] aligner and achieves a 3.5x speedup over the CPU implementation. BarraCUDA is a CUDA implementation of the BWA [9] aligner that achieves a 10x-15x speedup over the CPU implementation. SOAP3 is a GPU implementation of the SOAP2 [19] aligner that claims a 7.5x speedup over BWA and a 20x speedup over the Bowtie [20] aligner[5].

There have also been a few FPGA-based aligners. These projects include an implementation of BWA by Convey Computers [22], a short read mapper implemented on the Pico Computing M-503 platform [23], and Tera-BLAST [24], implemented on the TimeLogic J-Series FPGA card. The Convey Computers alignment accelerator was implemented on two Convey HC-2^{ex} servers, each with 4 Xilinx Virtex-6 LX760 FPGAs. It had a speedup of 10x to 20x while also providing energy savings of 81%. The Pico Computing mapper was implemented on 8 Xilinx Virtex-6 LX240T FPGAs and gave a speedup of 250x over BFAST [25] and 31x when compared to Bowtie. Tera-BLAST is an FPGA implementation of the BLAST [26] aligner and was implemented with TimeLogic's proprietary J-Series FPGA card.[6] It also utilizes 32 CPU cores, and achieves a 27x speedup over an unaccelerated 32 core implementation, 41.4x if two J-Series cards are used.

Hardware acceleration of variant calling, on the other hand, has not been explored to the same extent. To date, there have been two GPU implementations and one FPGA implementation of variant calling. The first was a GPU-based implementation of genome-wide association studies[7][28]. This work achieved a speedup of 17.7x to 25.7x over the CPU-based algorithm. However, this implementation uses a χ^2 test algorithm that is not well suited for NGS data. The other GPU implementation is GSNP [29], a GPU implementation of the SOAPsnp variant caller [30]. This work was able to achieve

[4]Short for Nvidia's Compute Unified Device Architecture, a popular GPGPU programming model.

[5]It should be noted here that Bowtie has since been supplanted by Bowtie 2 [21].

[6]It is unclear how many FPGA cores are on the card, but it uses "the latest Xilinx FPGA chips" [27].

[7]These studies search for links between genetic variants and phenotypic expressions such as diseases.

a 42x to 50x speedup over SOAPsnp. The only FPGA implementation of variant calling is an upcoming accelerator for the Genome Analysis Toolkit (GATK) [31] on the Convey Computers HC-2, which has reportedly given speedups of 3.7x to 13x [32].

CHAPTER 3

ANALYSIS OF SAMTOOLS

This chapter provides a performance analysis of the SAMtools variant caller, along with a summary of the major steps of the algorithm.

3.1 SAMtools

SAMtools [33] is a C library and software package that supports viewing and manipulating genome alignments in the SAM/BAM file format [34] along with a variant caller that can detect SNPs and short indels.

3.1.1 Variant Calling Command Line

The recommended command line for the SAMtools variant caller is as follows:

```
# samtools mpileup -uf ref.fa s0.bam | bcftools call -O b -v -c - > var.bcf
```

In the first command (`samtools mpileup`), the `-u` flag indicates that uncompressed output should be generated, the `-f` flag indicates the filename of the FASTA [35] file containing the indexed reference genome (`ref.fa`), and `s0.bam` is the name of the file containing the position sorted aligned reads. The results of this command are sent to STDOUT, which is then redirected to the second command (`bcftools call`).

The flags for `bcftools call` are: compressed binary output format (`-O b`), only output variants (`-v`), and use the consensus caller (`-c`)[1]. The input file is specified as `-`, indicating that the program should get its input from STDIN.

3.2 Architecture of `samtools`

```
1  while not at end of aligned reads do
2      bam_mplp_auto();
3      group_smpl();
4      forall the samples do
5          bcf_call_glfgen();
6      bcf_call_combine();
7      bcf_clear();
8      bcf_call2bcf();
9      bcf_write();
10     if not only calling SNPs then
11         if we have not reached maximum depth for indels then
12             indel = bcf_call_gap_prep();
13             if indel >= 0 then
14                 forall the samples do
15                     bcf_call_glfgen();
16                 bcf_call_combine();
17                 if this is a valid indel then
18                     bcf_clear();
19                     bcf_call2bcf();
20                     bcf_write();
```

Algorithm 1: Main loop of `mpileup`. The routine has been simplified to omit error-checking and initialization code.

The `samtools` program has 21 different commands, one of which is `mpileup`. As a result, the `main` function does little work other than checking for a valid

[1] An alternative to this option is the `-m` flag, which enables the multiallelic caller. However, this caller is considered experimental and is not investigated in this work.

command and calling the relevant function. In the case of `mpileup`, it calls `bam_mpileup`. This function acts as a `main`-like function, initializing data structures and parsing command line arguments. It then calls `mpileup`, which is responsible for 94.18% of total run time. After opening all input and output files, allocating memory, and initializing data structures, it enters the main loop listed in Algorithm 1. The purposes of the functions in this loop will be discussed in the next few sections. Run times reported for these functions are based on the use of the HG00150 data set [36] as an input and execution on a server with four AMD Opteron 6272 processors and 256GB of RAM (see Section 3.4 for details).

3.2.1 `bam_mplp_auto`

This function iterates through each position in the reference genome and returns the bases aligned to that position from the input BAM files. It is responsible for 3.03% of total run time.

3.2.2 `group_smpl`

One of the features of SAMtools is the ability to call variants on several different samples simultaneously. However, the input reads are sorted by position in the reference genome, not by sample. This function divides the reads for each sample into separate arrays. This operation involves a number of hash table lookups and memory reallocations, and requires 15.27% of total run time.

3.2.3 `bcf_call_glfgen`

This function is the second most expensive, taking 16.92% of total run time. It performs the following tasks:

- Extract the bases from the BAM data structure, along with their base quality scores, mapping quality scores, and minimum distance. The base quality score is the quality score given by the sequencer. The mapping quality score, on the other hand, is a similar score provided by the alignment tool as a measure of the alignment quality. The minimum distance is a measure of how close the base is to the beginning or end of its read.[2]
- Sum the quality scores for each base type (A, C, G, T, or other) observed in the reads.
- Count the number of reference genome matches and mismatches for forward and reverse[3] reads.
- Find the sums of the base quality scores, mapping quality scores, and minimum distances along with their squares for all reference genome matches and mismatches. The results of the previous item and this one are referred to henceforth as the *annotations*.
- Histogram the number of reference genome matches and mismatches at each position along the read.
- Histogram the base quality and mapping quality scores.
- Calculate the Phred-scaled likelihoods[4] for each genotype. This calculation is done with the `errmod_cal` function, which is an implementation of Equation 2 in [13] that accounts for error dependencies.

[2]This information is useful because bases tend to be less reliable when they are near the beginning or end of the read

[3]In *paired-end* sequencing, each read is built twice, once in the *forward* direction and once in the *reverse* direction.

[4]The Phred-scaled likelihood is the log-scaled error probability, calculated as $-10 \log_{10} P$ [37].

3.2.4 `bcf_call_combine`

This function takes 23.63% of total run time, the most of any of the functions called by `mpileup`. It performs the following tasks:

- Translate the 4-bit reference base to a 3-bit representation (A = 0, C = 1, G = 2, T = 3, all other bases[5] = 4).
- Sum the per-base type quality score sums calculated in `bcf_call_glfgen` across all samples, then sum the results across all base types. Summing across the samples allows for population-level trends to be used in calling variants, while the overall sum is used later to normalize the per-base sums.
- Encode the quality score sums with their respective bases in the lowest 3 bits, then sort in descending order. This step in effect sorts the possible alleles by their cumulative quality score.
- Determine if any base types are "unseen" (i.e., have a cumulative quality score of 0).
- Find the possible genotypes that can be constructed from the observed bases in descending order of quality score.
- Translate the Phred likelihoods for each sample so that their minimum value is zero.
- Calculate the segregation based metric [38], which is the likelihood of a suspected variant being a true variant or noise based on its appearance among all of the samples being tested.
- Combine the annotations across all samples.
- Find the value of the Mann-Whitney U test [39] for the histograms found in `bcf_call_glfgen`. This calculation is done recursively when the sample size of both the reference and alternate histograms is between 3 and 7 (inclusive). A linear approximation is used if the sample

[5]These "other" bases represent a number of cases where the result from the sequencer was indeterminate. See the FASTA specification [35] for more details.

size of either is 2, and a normal approximation is used when either of the sample sizes is greater than or equal to 8. A sample size of 1 for either sample results in a flat probability of 1.0.

- Calculate the Variant Distance Bias [40] for the average distance from the average position of variant bases.

3.2.5 `bcf_clear`

This function simply frees the memory allocated in a `bcf1_t` data type (this type represents a single record in a BCF[6] file) and reinitializes its member values to zero so that it can be reused in every iteration of the main loop. It requires only 1.33% of total run time.

3.2.6 `bcf_call2bcf`

This function converts the results of `bcf_call_combine` to the `bcf1_t` type. Its main operations are string formatting, memory allocation, and copying between arrays. It requires 16.86% of total run time.

3.2.7 `bcf_write`

This function writes a `bcf1_t` type to a file. Its main operations are memory copying, gzip compression,[7] and file writes. It requires 7.72% of total run time.

[6] BCF is the binary version [41] of the VCF format [42].

[7] Using the libz library. This option is generally not used if the output is being piped directly into bcftools.

3.2.8 bcf_call_gap_prep

```
REF:      gacgctcctctagt----tccttccgccaggagtacacgg
SMP:      gacgctcctctagtTACTtccttccgccaggagtacacgg
001:      gacgctcctctagt****tActtcc
002:       acgctcctctagt****tActtccTTc
003:           tcctctagtTACTtccttccgccaggag
004:               TtaCt****tccttccgccaggagtaca
005:                 aCt****tccttccgccaggagtacacg
006:                  Ct****tccttccgccaggagtacacgg
```

Figure 3.1: Example of how a short indel (`TACT`) can be mapped as multiple SNPs. The `REF` line is the reference, the `SMP` line is the actual sample, and the `001-006` lines are the reads. Capital letters indicate a difference from the reference, hyphens (`-`) indicate that the position is not present in the reference, and asterisks (`*`) indicate the position is not present in the alignment of that read.

It is possible for a read containing a short indel to be misaligned as a small number of SNPs during pairwise alignment (see Figure 3.1 for an example). This can occur when the cost of an insertion in the alignment algorithm is much higher than that of consecutive SNPs [43]. Li developed a method to account for this ambiguity with a hidden Markov model that computes an additional quality value called the base alignment quality [44]. This function identifies whether an indel may be present within a window of 100 bp, then computes the forward algorithm in [44] to determine the likelihood of each of the alignments within the window given the reference sequence. If an indel has the highest likelihood, then a value of 0 is returned. Otherwise, a value of -1 is returned. This function takes 3.25% of total run time.

3.3 Architecture of `bcftools`

```
1  while not at end of input do
2      bcf_sr_next_line();
3      if only calling on subset of samples then
4          bcf_subset();
5      bcf_unpack();
6      if only calling variants and not variant then
7          continue;
8      if only calling indels and not indel then
9          continue;
10     if not calling indels and indel then
11         continue;
12     if only output {A, C, G, T} and base ∉ {A, C, G, T} then
13         continue;
14     bcf_unpack();
15     ccall();
16     if only output variants then
17         if not a variant then
18             continue;
19     bcf_write();
```

Algorithm 2: Main loop of `bcftools`. This algorithm has been simplified to omit error-checking and initialization code.

The architecture of `bcftools` is similar to `samtools` in that it has 13 different commands, each of which has its own `main`-like function. For the `call` command, that function is `main_vcfcall`, reproduced in Algorithm 2. Unlike the `mpileup` function, this function only has four children.[8] The purposes of these functions are listed in the next sections.

[8]The profiling configuration operates across all samples, so no calls are made to `bcf_subset`.

3.3.1 `bcf_sr_next_line`

Just as `bam_mplp_auto` iterates through a BAM file, this function iterates through a BCF file and returns the data for each record, decompressing it if necessary. This function requires 91.49% of total run time.

3.3.2 `ccall`

This function calculates a number of statistics and calls the variants. It performs the following operations:

- Estimate the reference allele frequency using either expectation maximization (Equation 5 in [13]) or maximum likelihood (Brent's method [45]), depending on speed of convergence.
- Estimate the genotype frequencies for all combinations of reference (R) and alternate (A) alleles (RR, RA, AA) using expectation maximization.
- Find the Hardy-Weinberg Equilibrium (HWE)[9] P-value. This value is a measure of how well the data follows the HWE.
- Divides the samples into two groups and determines their reference allele frequencies and the first-degree P-value for these frequencies.
- Finds the second-degree P-value for the genotype frequencies calculated in the second item.
- Estimates the allele frequency spectrum (Equation 21 in [13]), folded variant probability, and the expected reference allele frequency.
- Estimates the equal-tail credible interval and computes the likelihood ratio test (Equation 8 in [13]).

[9]This law states that, in the absence of evolutionary pressures (e.g. mutation, selection, etc.), the allele and genotype frequencies will stay constant in a given population [46].

- Using the expected reference allele frequency, computes the prior probabilities for each genotype (RR, RA, AA) and multiplies by the likelihoods calculated in `samtools mpileup` to find all possible values of the right-hand side of Equation 2.2. These values are then used to find the genotype maximizing the posterior probability.
- Writes the statistics and the final variant calls to the output BAM file.

This function requires 4.04% of total run time.

3.3.3 `bcf_unpack`

This function simply unpacks all or part of a BCF record into a `bcf_dec_t` type so that the data can be easily accessed. This function requires 1.85% of total run time.

3.3.4 `bcf_write`

This function is the same `bcf_write` called by `mpileup` (Section 3.2.7). It requires 0.46% of total run time.

3.4 Performance Analysis

The performance of SAMtools was profiled using the Linux perf tool,[10] using the command line options given in Section 3.1.1. The test data consisted of the full mapped human genome from the HG00150 data set in the 1000 Genomes Project [36]. The profiling took place on a server with four AMD

[10] `https://perf.wiki.kernel.org/index.php/Main_Page`

Table 3.1: `mpileup` Profiling Results

Function[a]	**Execution Time (% of Total)**
bcf_call_combine	23.63
bcf_call_glfgen	16.92
bcf_call2bcf	16.86
group_smpl	15.27
bcf_write	7.72
bcf_call_gap_prep	3.25
bam_mplp_auto	3.03
___memset_sse2	2.52
_int_free	1.54
bcf_clear	1.33
free	0.67
bam_aux_get	0.17

[a]Only functions that are direct children of `mpileup` in the call graph are considered. See Figure 3.2 for the full call graph, and Figure 3.3 for a reduced call graph focused on the above functions.

Opteron 6272 processors (with 8 cores each) and 256GB of RAM. The run time percentages in the profiling results that follow are with respect to the individual run time of each of the two sub-commands (`samtools mpileup` and `bcftools call`), not the command line as a whole.

3.4.1 Profiling Results of `samtools mpileup`

The profiling results for `samtools mpileup` indicated that the function `mpileup` in the file bam_plcmd.c is responsible for 94.18% of total run time. This function splits its run time among several callee functions. The results for these functions are shown in Table 3.1, and their purposes are discussed in greater detail in Section 3.2.

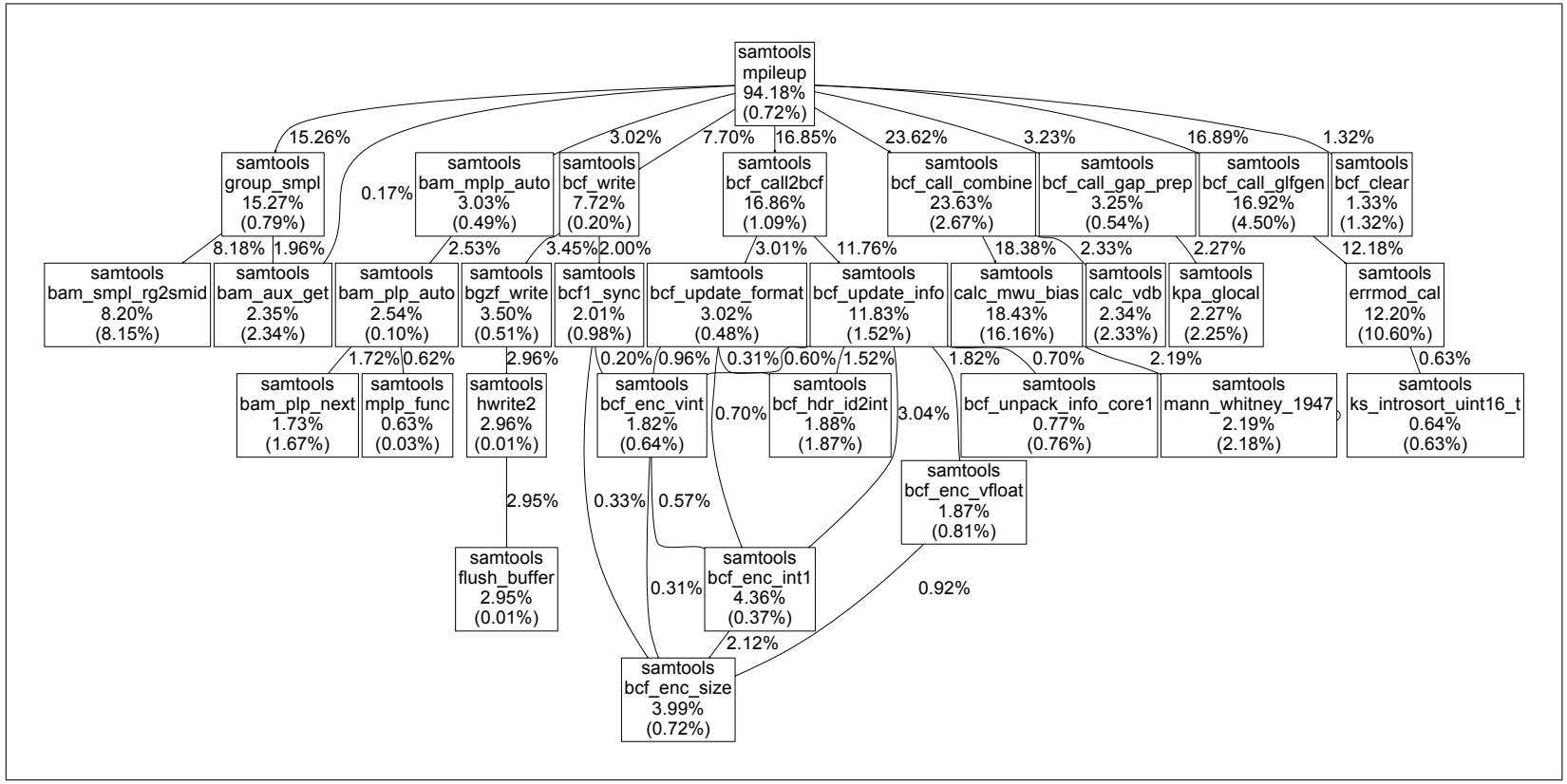

Figure 3.2: Call graph of `samtools mpileup`. Nodes corresponding to kernel and libc functions have been removed to improve readability.

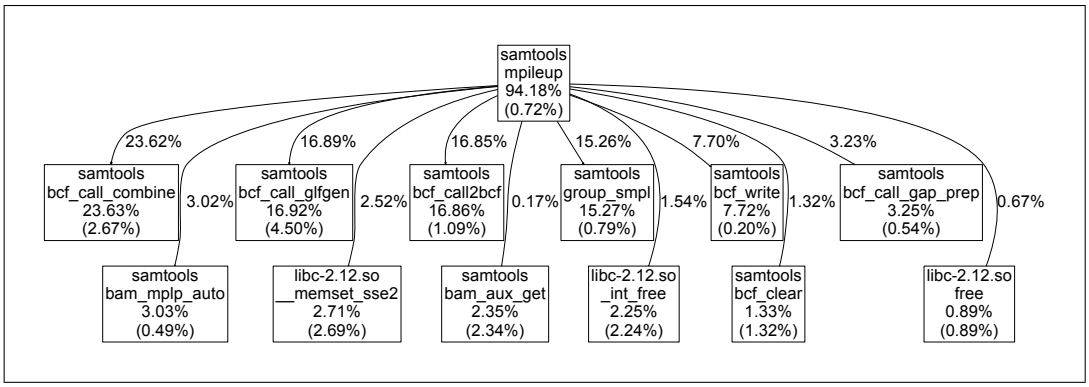

Figure 3.3: Reduced call graph of `samtools mpileup` showing only the `mpileup` function and its immediate callees.

3.4.2 Profiling Results of `bcftools call`

The profiling results for `bcftools call` were similar in that the function `main_vcfcall` in the file vcfcall.c took the bulk of the execution time (99.46%). Table 3.2 shows the profiling results of its immediate callees, which are discussed in Section 3.3.

3.4.3 Relative Performance

Because `samtools mpileup` and `bcftools call` are run as two separate processes, it is difficult to determine exactly how much each process contributes to overall run time.

To determine the relative run time of the two tools, `samtools mpileup` was first run alone, with its results piped to /dev/null. Then, the full command was timed. The difference between the two times should be the contribution of `bcftools call`. The wall clock time measurements listed in Table 3.3 were gathered using the GNU `time` utility.

These results indicate that the effects of `bcftools call` on total run time are minor. Assuming that the `samtools mpileup` process required approximately the same amount of time as when it was run alone, `bcftools call`

Table 3.2: `main_vcfcall` Profiling Results

Function[a]	**Execution Time (% of Total)**
bcf_sr_next_line	91.49
ccall	4.04
bcf_unpack	1.85
bcf_write	0.46

[a]Only functions that are direct children of `main_vcfcall` in the call graph are considered. See Figure 3.4 for the full call graph, and Figure 3.5 for a reduced call graph focused on the above functions.

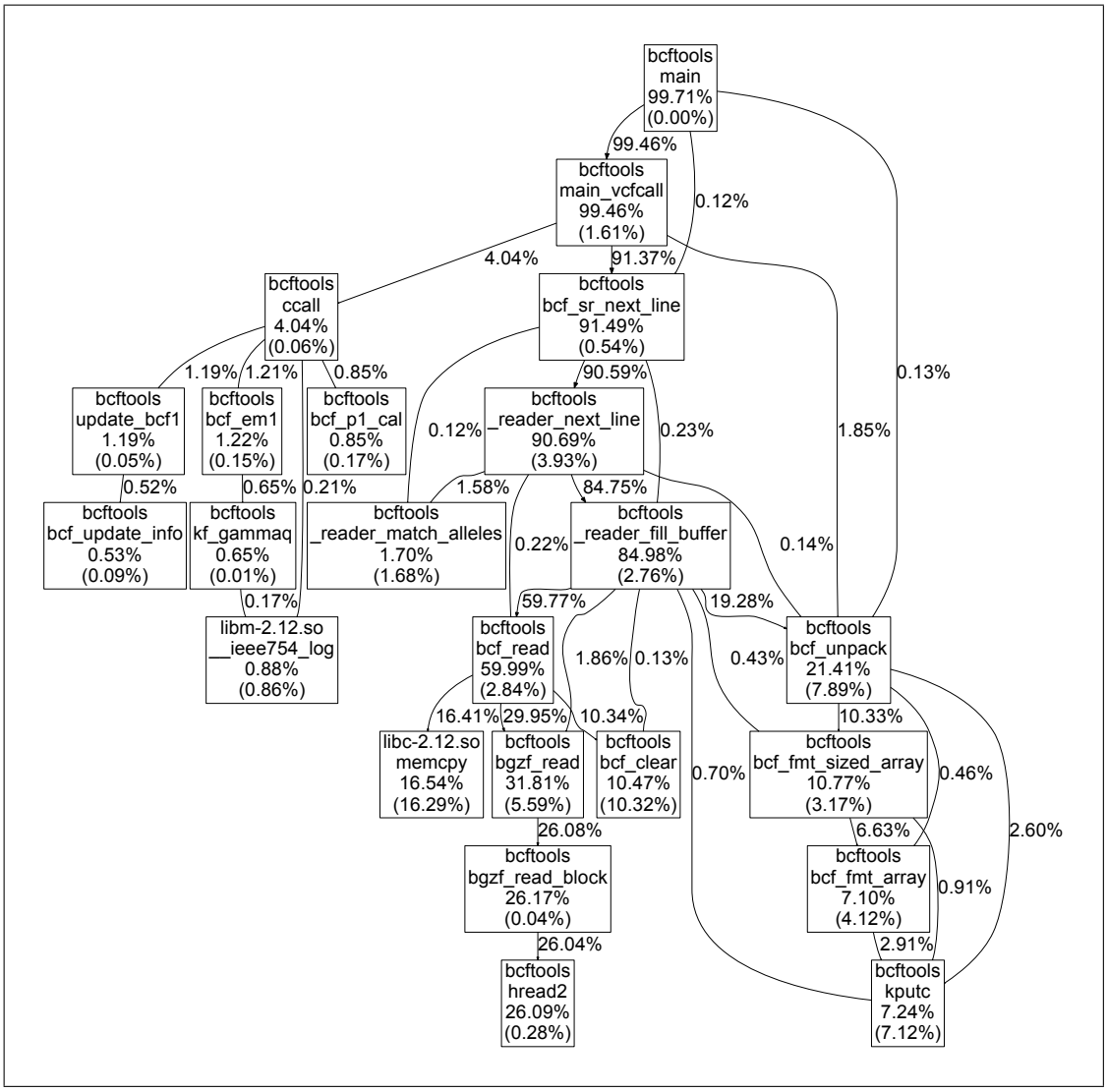

Figure 3.4: Call graph of `bcftools call`. Nodes corresponding to kernel and libc functions have been removed to improve readability.

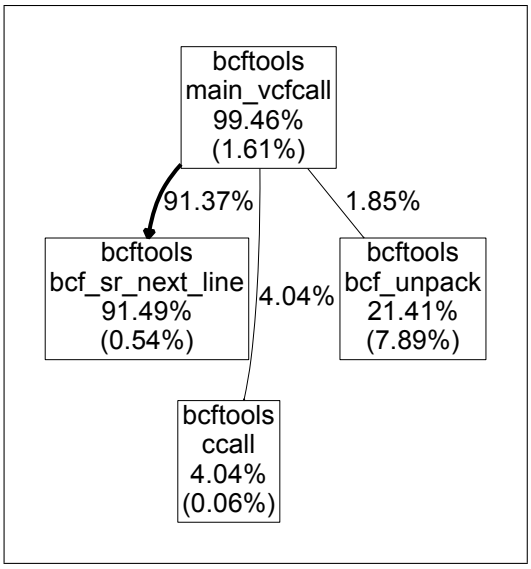

Figure 3.5: Reduced call graph of `bcftools call` showing only the `main_vcfcall` function and its immediate callees.

Table 3.3: Per-command Wall Clock Time

Command	**Time (min)**
`samtools`	331.7
`samtools` + `bcftools`	352.2

only added 20.5 minutes to the total run time, accounting for 5.82% of total run time.

CHAPTER 4

IMPLEMENTATION

This chapter presents the software and hardware implementations of this project. The software implementation optimizes SAMtools to reduce its I/O overhead. The functions `bcf_call_glfgen` and `bcf_call_combine` are also implemented on an FPGA to further improve performance.

4.1 Software

As the profiling results demonstrate, both `samtools mpileup` and `bcftools call` spend a considerable amount of time on the intermediate piped I/O (24.58% of `samtools mpileup` and 93.34% of `bcftools call`). While the authors designed SAMtools this way for the sake of high flexibility [13], the intermediate file is rarely used in practice [47]. Therefore, it is clear that significant speedup could be achieved by combining the two tools and eliminating the intermediate pipe.

This change is implemented as a new program named `mpileup_call`. It combines the main loops of the `mpileup` and `main_vcfcall` functions while simplifying `bcf_call2bcf` and eliminating calls to `bcf_write`, `bcf_sr_next_line`, and `bcf_unpack`.

```
1  while not at end of aligned reads do
2      bam_mplp_auto();
3      if number of samples = 1 then
4          my_group_smpl();
5      else
6          group_smpl();
7      if not only calling indels then
8          forall the samples do
9              bcf_call_glfgen();
10         bcf_call_combine();
11         call_convert();
12         if not (only output variants and not a variant) then
13             bcf_clear();
14             my_bcf_call2bcf();
15             my_ccall();
16             if not (only output variants and not a variant) then
17                 bcf_write();
18     if not only calling SNPs then
19         if we have not reached maximum depth for indels then
20             bcf_call_gap_prep();
21             if there is an indel at this position then
22                 forall the samples do
23                     bcf_call_glfgen();
24                 bcf_call_combine();
25                 if this is a valid indel then
26                     call_convert();
27                     if not (only output variants and not a variant)
                       then
28                         bcf_clear();
29                         my_bcf_call2bcf();
30                         my_ccall();
31                         if not (only output variants and not a variant)
                           then
32                             bcf_write();
```

Algorithm 3: Main loop of `mpileup_call`.

4.1.1 Main Loop

Structurally, this loop (Algorithm 3) is similar to the main loop of `mpileup`. The main difference is that the SNP calling step (the first set of calls to `bcf_call_glfgen` and `bcf_call_combine`), has been made conditional on whether the tool should only call indels. This change was necessary because `bcftools call` has options to individually disable SNP and indel calling, while `samtools mpileup` always performs SNP calling. Performing SNP calling when the user has disabled it in the `bcftools` options would be wasted work.

To bridge the `samtools` and `bcftools` parts of the program, a new function (`call_convert`) has been added at lines 11 and 26. This function converts from the `bcf_call_t` type output by `bcf_call_combine` to the `call_t` type expected by `ccall`. This replaces calls to `bcf_write` and `bcf_sr_next_line` with a much simpler and faster copy operation. The source of `call_convert` is reproduced in Listing 4.1.

Another change is the replacement of `group_smpl` with `my_group_smpl` if there is only one sample. The expensive hash table operations in `group_smpl` can be completely removed, since all reads map to the same sample.

The next change replaces `bcf_call2bcf` with `my_bcf_call2bcf` at lines 14 and 28. This function is largely the same, except it removes the lines that create the I16 and QS fields in the output BCF, since these fields are later removed by `bcftools call`. The information in those fields is instead copied directly by `call_convert`.

Similarly, `ccall` is replaced by `my_ccall` at lines 15 and 29. This function and its callees (`bcf_em1` and `bcf_p1_cal`) are altered to remove any I/O operations from the intermediate BCF file. These operations are replaced by either setting those values in `call_convert` or passing them as additional parameters to those functions. The function `update_bcf1` is also replaced by `my_update_bcf1`, which omits the lines that access and remove the I16 and QS fields mentioned earlier.

Listing 4.1: Source of `call_convert`.

```
char call_convert(bcf_call_t* in, call_t* out,
  int* num_alleles, int* num_samples)
{
  char is_snp;
  int i, nals = 1;
  out->hdr = in->bcf_hdr;
  if(out->anno16 == NULL)
    out->anno16 = (float*)malloc(16*sizeof(float));
  for(i = 0; i < 16; i++) {
    out->anno16[i] = in->anno[i];
  }
  is_snp = (in->ori_ref >= 0);
  if(!is_snp) {
    for (i=1; i<4; i++) {
      if (in->a[i] < 0) break;
      nals++;
    }
  }
  else {
    for (i=1; i<5; i++) {
      if (in->a[i] < 0) break;
      nals++;
    }
  }
  out->PLs = in->PL;
  int ngts = nals*(nals+1)/2;
  out->nPLs = out->mPLs = ngts * in->n;
  *num_alleles = nals;
  *num_samples = in->n;
  return is_snp;
}
```

4.2 Other Optimizations

Other optimizations were made to the `bcf_callret1_t` (Listing 4.2) and `bcf_call_t` (Listing 4.3) data types. In the original implementation of `bcf_callret1_t`, the `qsum` member is a `float` type, and the `anno` mem-

ber is an array of `double` types. However, these members are never used for floating point operations. The `qsum` member is an integer sum created in `bcf_call_glfgen`. The `anno` member is created in the same function and contains several different integer sums. As Listing 4.2 shows, these members have been changed to `int` and `unsigned long` types. These changes both improve the performance of `bcf_call_glfgen` and reduce the amount of floating point math required in `bcf_call_combine`.

Listing 4.2: Optimized definition of `bcf_callret_t` data type.

```
typedef struct {
  uint32_t ori_depth;
  unsigned int depth, n_supp, mq0;
  int qsum[4];
  unsigned long anno[16];
  float p[25];
} bcf_callret1_t;
```

Listing 4.3: Optimized definition of `bcf_call_t` data type.

```
typedef struct {
  int tid, pos;
  bcf_hdr_t *bcf_hdr;
  int a[5];
  float qsum[5];
  int n, n_alleles, shift, ori_ref, unseen;
  int n_supp;
  unsigned long anno[16];
  unsigned int depth, ori_depth, mq0;
  uint32_t *PL, *DP, *DV;
  float vdb;
  float mwu_pos, mwu_mq, mwu_bq, mwu_mqs;
  float seg_bias;
  kstring_t tmp;
} bcf_call_t;
```

The `bcf_call_t` data type was optimized by changing its `anno` member to a long integer, since it is calculated by summing the `anno` members from several `bcf_callret1_t` types. The new definition also omits the members `mwu_pos_cdf`, `mwu_mq_cdf`, `mwu_bq_cdf`, and `mwu_mqs_cdf`. These members are only used if SAMtools is compiled with `CDF_MWU_TESTS` enabled, but it is

disabled by default. These members are therefore unnecessary and removing them reduces memory usage.

4.3 Hardware

The hardware accelerator is implemented in a mix of Verilog and SystemVerilog, targeted for the Terasic DE5 development board, which features an Altera Stratix V GX A7 FPGA. This design implements `bcf_call_glfgen`, `bcf_call_combine`, and most of their callees in hardware. The accelerator is used to replace the calls to `bcf_call_glfgen` and `bcf_call_combine` at lines 8-10 and 22-24 in Algorithm 3. In order to simplify the design and reduce area and memory requirements, this design currently only supports variant calling for a single sample.

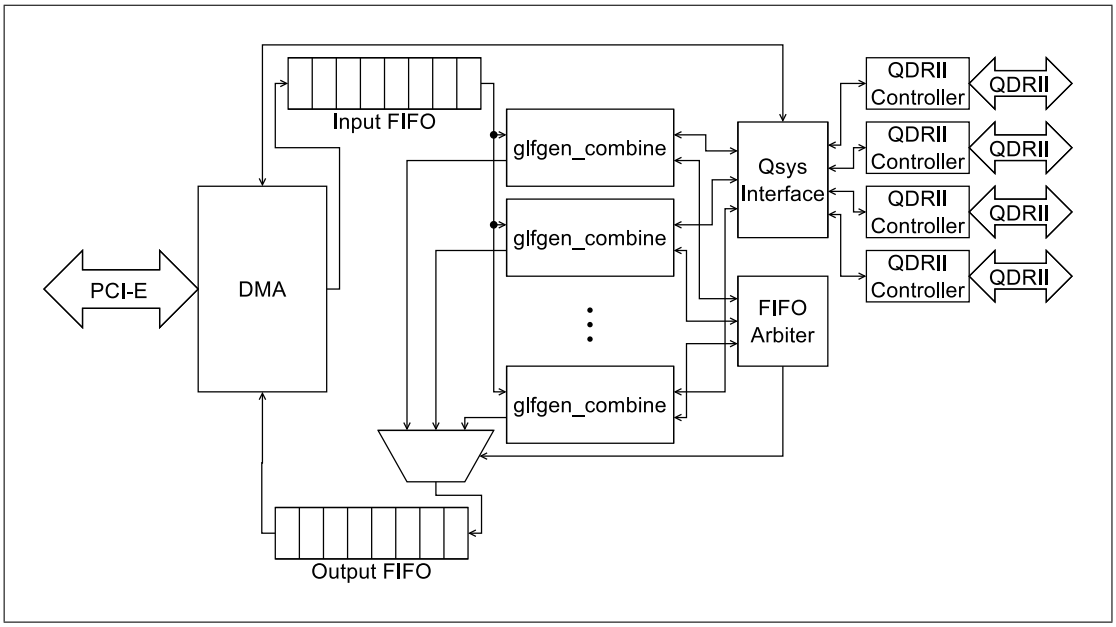

Figure 4.1: Top-level architecture of the accelerator.

4.3.1 Top-Level Architecture

The architecture is modular and parameterized, and allows for an arbitrary number of processing units (`glf_combine`) to be instantiated, limited only by timing and area requirements. There are two on-chip First In, First Out (FIFO) memory blocks created using Altera FIFO IP, one for input packets and one for output packets (see Section 4.3.2.3 for a description of the packets). The processing units are given access to the FIFOs by the FIFO arbiter, using a fixed round-robin policy. A diagram of the top-level architecture is shown in Figure 4.1.

4.3.2 Host <-> Accelerator Interface

The FPGA communicates with the host computer via a PCI Express (PCIe) Gen 3.0 x8 interface. This interface is implemented using a reference design provided by Altera [48]. This design provides a Direct Memory Access (DMA) engine that transfers data from the PCIe interface to an on-chip FIFO with an input width of 256 bits, an output width of 128 bits, and a depth of 32768 entries. The lower output width was chosen in order to match the size of the data types being sent (see Figure 4.2). Results from the accelerator are written into another on-chip FIFO with width of 256 bits and a depth of 1024 entries. The Linux driver provided for this interface by Altera was modified to achieve higher performance.

4.3.2.1 Driver

The driver maps two DMA buffers, one for reads and the other for writes. These buffers are can be accessed from user space using `read()` and `write()` calls or by mapping them with `mmap()` and accessing them directly. Memory mapping gives better performance because it avoids expensive copy operations between user space and the kernel. Reads and writes are initiated with

`ioctl()` calls. It is also possible to perform simultaneous read/write operations to maximize bandwidth utilization. Other `ioctl()` commands provide interfaces to configure the number of packets to be processed by each processing unit, query the utilization of the input FIFO, and set, clear, and wait on control signals to and from the FPGA. If a transfer would be larger than the size of the input FIFO, then the driver breaks the transfer up into multiple transfers of smaller chunks. After writing a chunk, it waits until the input FIFO has enough entries free and writes the next chunk. This process repeats until all data are transferred.

4.3.2.2 Control Signals

The Altera DMA controller does not provide an interface to allow user logic to trigger interrupts[1] to the host. Instead, control signals between the host and the FPGA are implemented with memory-mapped parallel I/O (PIO) modules accessed over the PCIe bus.

There are four control signals, two from the host to the FPGA, and two from the FPGA to the host. The host to FPGA control signals are:

- ALTERA_IRQ_RESET: Resets the processing units. FIFOs, and FIFO arbiter.
- ALTERA_IRQ_LATCH_NUM_PER_INSTANCE: Instructs the user logic to latch the value in the PIO module containing the number of packets to process per instance (called `num_per_instance`).

The FPGA to host control signals are:

- HOST_IRQ_DONE: Signals that the FPGA has finished processing all data and has results ready.

[1] PCI Express uses Message Signaled Interrupts (MSI), which are signaled with special bus transactions rather than dedicated interrupt pins. Up to 32 different interrupts can be allocated. PCIe 3.0 also provides MSI-X, which allows for up to 2048 interrupts.

- HOST_IRQ_ACK_NUM_PER_INSTANCE: Signals that the FPGA has successfully latched `num_per_instance`.

In addition, there are two PIO modules that operate as control and status registers. One is a read-only status register that reports the number of entries in use in the input FIFO so that the host does not overflow the FIFO.

The other PIO module is `num_per_instance`, which is written by the host to indicate how many packets each processing unit should expect to process per transaction. Ensuring that each processing unit processes the same number of packets simplifies many aspects of the hardware design, including the arbiter and determining when to signal the host that computation has finished. It also places a hard upper limit on the size of the output FIFO, preventing wasteful over provisioning of memory blocks. Finally, it also reduces the complexity of the user space software that interacts with the FPGA while allowing for data transfers and host-side processing to be easily overlapped.

4.3.2.3 User Space Interface

In order to match the FIFO data width, the arguments to `bcf_call_glfgen` are first marshaled into a packet of `glf_header` and `glf_packed` data types (Listings 4.4 and 4.5). Figure 4.2 shows how the packet maps onto the on-chip FIFO.

The `glf_header` data type contains the reference base (`ref_base`) at the current position in the alignment along with how many bases are present in the alignment at that position (`n`). The `valid` field is a status bit used to indicate whether the header is valid. The `last` field is always set to 0. Figure 4.3 shows the memory layout of a `glf_header`.

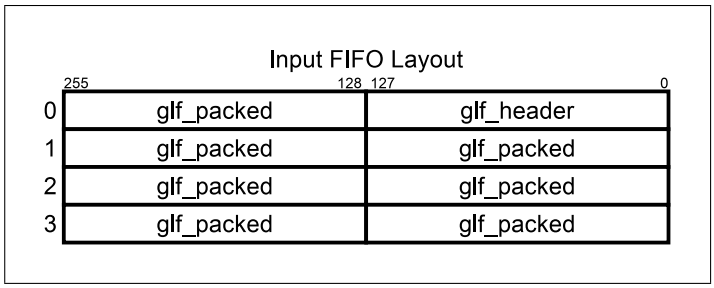

Figure 4.2: Example of how the data sent to the FPGA maps to the input FIFO. Each `glf_header` and `glf_packed` requires 16 bytes, or half of one FIFO entry.

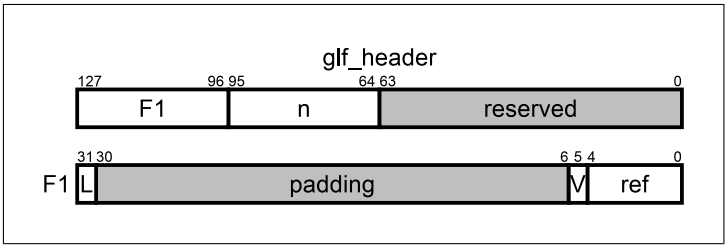

Figure 4.3: Memory layout of a `glf_header` structure. The area marked "reserved" corresponds to the `packed_data` pointer, which is only used by the host. The "V" and "L" fields correspond to the members `valid` and `last`.

Listing 4.4: Definition of `glf_header` data type.

```
typedef struct {
    glf_packed* packed_data;
    int n;
    int32_t ref_base:5, valid:1, pad:25, last:1;
} glf_header;
```

The `glf_header` is followed by one or more `glf_packed` types, one for each read in the alignment at the current position. The `glf_packed` type (see Figure 4.4 for how it maps to memory) contains information extracted from the input BAM files, including the read direction (`is_rev`), base type (`base`), indel-related flags and quality scores (`aux`), position within the read (`qpos`), minimum distance from the beginning or end of the read (`min_dist`), position relative to the aligned part of the read (`epos`), mapping quality (`core_qual`), and base quality (`base_qual`). The `load_array` and `last` members are hints to the accelerator. If `load_array` is set to 1, that base

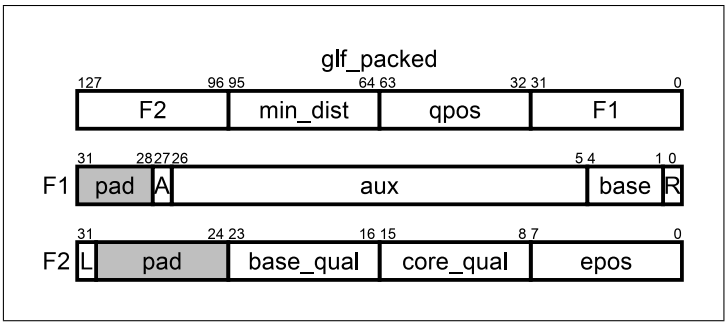

Figure 4.4: Memory layout of a `glf_packed` structure. The "R", "A", and "L" fields correspond to the members `is_rev`, `load_array`, and `last`.

is considered in the `errmod_cal` calculations. Otherwise, it is ignored. If `last` is set to 1, it indicates that the rest of the data in the FIFO is empty padding and instructs the accelerator to flush the FIFO. This operation is necessary because the FIFO is written 256 bits at a time, so incomplete 256 bit words must be padded with zeroes. If there are an even number of `glf_packed` types in the array sent to the FIFO, the resulting array has an odd number of 128 bit words (because the `glf_header` type is also 128 bits wide). Padding the array with zeros to make it a multiple of 256 bits avoids any alignment issues.

Listing 4.5: Definition of `glf_packed` data type.

```
typedef struct {
    uint32_t is_rev:1, base:4, aux:22, load_array:1, pad:4;
    int32_t qpos, min_dist;
    uint32_t epos:8, core_qual:8, base_qual:8, pad2:7, last:1;
} glf_packed;
```

Each processing element returns its results in a `pcie_packet_in` data type (Listing 4.6). Figure 4.5 shows how the `pcie_packet_in` data type maps on to the output FIFO. The `pcie_packet_in` type is a further optimized version of the `bcf_call_t` in Listing 4.3, and most of its members map directly to the original version.

Listing 4.6: Definition of `pcie_packet_in` data type.

```
typedef struct
{
  uint32_t call_pl[15];
  float qsum[5];
  int n, n_alleles, shift, ori_ref, unseen;
  unsigned int depth, ori_depth, mq0;
  unsigned long bca_anno[2];
  unsigned long anno[16];
  int32_t vdb_flag:1, pos_flag:1, mq_flag:1, bq_flag:1,
    mqs_flag:1, a0:3, a1:3, a2:3, a3:3, a4:3, result_code:12;
  uint32_t DP, DV;
  float vdb;
  double mwu_pos, mwu_mq, mwu_bq, mwu_mqs;
  uint64_t pad2[2];
} pcie_packet_in;
```

The differences in `pcie_packet_in` are:

- The member order has been changed to allow the processing elements to write the data to the FIFO in the order it is generated.
- The `PL` pointer in the original structure has been replaced with the statically allocated `call_pl` array to simplify memory operations. The 15 member size is valid for a single sample.
- The `bca_anno` array has been added because `calc_SegBias` is not implemented in hardware. This computation is instead performed as a post-processing step.
- The `a` array has been converted into five members of a bit field, each three bits wide (`a0`, `a1`, `a2`, `a3`, `a4`). This optimization is made possible by the fact that the members of `a` can only take the values -1, 0, 1, 2, 3, or 4.
- Additional status flags are also located in the bit field containing the `a` values. These flags are used to indicate whether other members need further post-processing in software. The `vdb_flag` bit corresponds to the `vdb` value, `pos_flag` corresponds to `mwu_pos`, `mq_flag` corresponds

to `mwu_mq`, `bq_flag` corresponds to `mwu_bq`, and `mqs_flag` corresponds to `mwu_mqs`.

- The last member of the bit field is `result_code`, which corresponds to the return code for the software version of `bcf_call_combine`. The function returns 0 if the results are valid or -1 if an error was encountered. This value is made 12 bits wide for alignment purposes.
- The `DP` and `DV` pointers have been replaced with single integers. This optimization is only valid for the single sample case.
- The `mwu_pos`, `mwu_mq`, `mwu_bq`, and `mwu_mqs` members have been changed to double types. They are converted to float types after post-processing.
- The `pad` member makes the data type 320 bytes so that it aligns properly in the output FIFO.

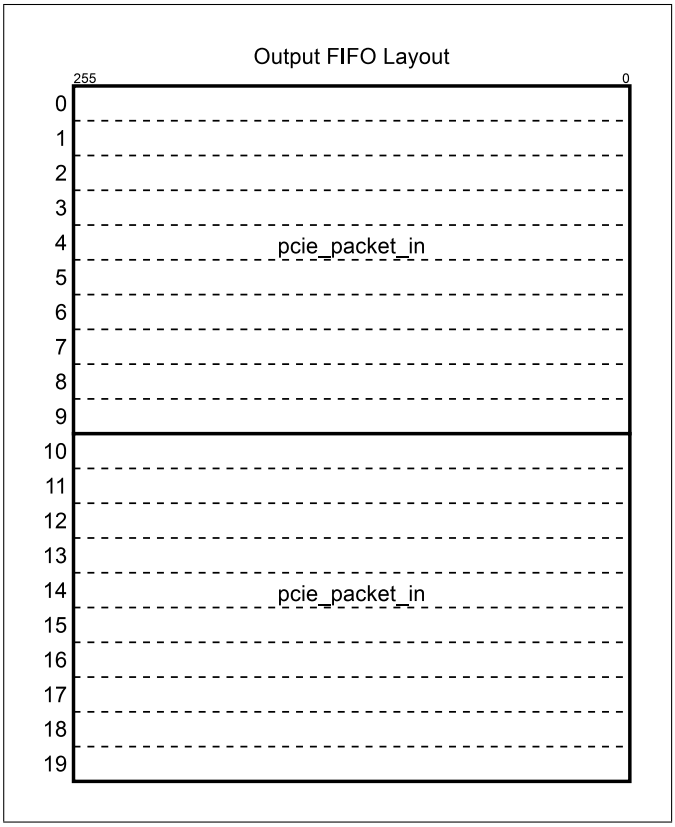

Figure 4.5: Example of how the data returned by the FPGA maps to the output FIFO. Each `pcie_packet_in` requires 320 bytes, or 10 entries in the FIFO.

If the FPGA is configured with N processing units, then a total of $N \cdot$ `num_per_instance` packets are sent at a time. In order to improve efficiency, the user space program uses simultaneous read/write transfers whenever possible. It also post-processes returned data and prepares the next set of packets to be sent while the FPGA is busy, effectively pipelining the transfers and processing in software. The depth of this software pipeline is configurable at compile time with the `IO_GRANULARITY` parameter. Empirical tests found that setting `IO_GRANULARITY` to 5 provides the best performance.

4.3.3 Hardware Accelerated `mpileup_call`

The hardware-accelerated version of `mpileup_call` is a separate program named `mpileup_call_accel`. The main loop for `mpileup_call_accel` (Algorithm 4) does not change much from the software-only version. The main differences are:

- The function `bam_mplp_auto` is replaced by `bam_mplp_auto_multi`. This new function fetches the aligned reads and reference genome data over a window of bases `NUM_PACKETS` long. Setting `NUM_PACKETS` to 300 provides the best performance.
- The function `group_smpl` is fully replaced by `my_group_smpl`, since only one sample is allowed.
- The calls to `bcf_call_glfgen` and `bcf_call_combine` are replaced by a single call to a new function named `glfgen_combine`. This function is discussed in more detail below.
- The order of the function calls has been altered. This change was necessary for the data written to the output BCF file to have the correct order.

The new function `glfgen_combine` (Algorithm 5) implements a software pipeline that overlaps host-FPGA data transfer with pre- and post-processing.

```
1  while not at end of aligned reads do
2      bam_mplp_auto_multi(NUM_PACKETS);
3      my_group_smpl();
4      if not only calling indels then
5          glfgen_combine(SNP);
6      if not only calling SNPs then
7          if we have not reached maximum depth for indels then
8              bcf_call_gap_prep();
9              glfgen_combine(INDEL);
10     forall the valid packets returned do
11         if this is a valid SNP then
12             if not only calling indels then
13                 call_convert();
14                 if not (only output variants and not a variant) then
15                     bcf_clear();
16                     my_bcf_call2bcf();
17                     my_ccall();
18                     if not (only output variants and not a variant)
                       then
19                         bcf_write();
20         if this is a valid indel then
21             call_convert();
22             if not (only output variants and not a variant) then
23                 bcf_clear();
24                 my_bcf_call2bcf();
25                 my_ccall();
26                 if not (only output variants and not a variant) then
27                     bcf_write();
```

Algorithm 4: Main loop of hardware accelerated `mpileup_call`.

It attempts to use simultaneous read/write transfers whenever possible to maximize bandwidth utilization. While the FPGA is working on a data set, `glfgen_combine` post-processes any data returned from the last iteration and prepares the next data set to be sent. It operates on `NUM_PACKETS/IO_GRANULARITY` bases at a time.

```
1  prepare_glf();
2  if there are valid packets to send then
3      start_write();
4  while not all packets sent to the accelerator do
5      prepare_glf();
6      if there are valid packets to send then
7          if there is an outstanding request then
8              wait_for_done();
9              start_simul();
10         else
11             start_write();
12     else if there is an outstanding request then
13         start_read();
14     if data was read back from the FPGA then
15         postprocess_bcf();
16 if there is an outstanding request then
17     start_read();
18     postprocess_bcf();
```

Algorithm 5: Algorithm for `glfgen_combine`.

Data preparation takes place in a function named `prepare_glf`. For each position within the current chunk of bases, the function calls `glf_pack`, which extracts data from a `bam_pileup1_t` type and initializes a `glf_packed` array in the memory mapped buffer provided by the driver. The `prepare_glf` function also ensures that the resulting set of packets is correctly aligned to a 256 bit boundary and inserts invalid headers to account for any packets that were filtered completely,

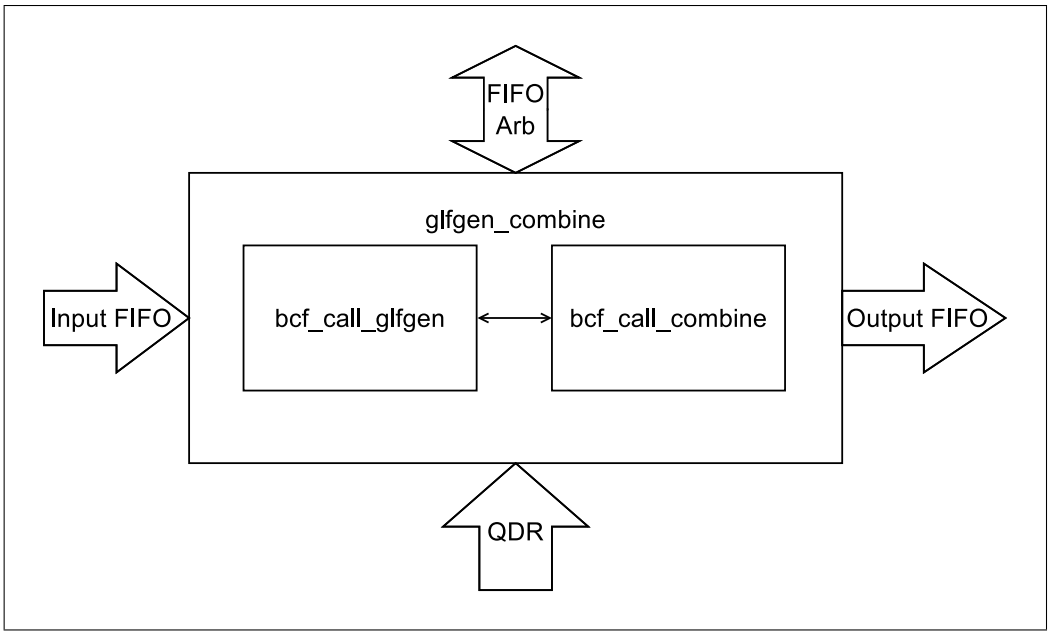

Figure 4.6: Architecture of a `glfgen_combine` processing unit.

Post-processing takes place in `postprocess_bcf`. This function copies data from the `pcie_packet_in` array returned by the accelerator to the `bcf_call_t` array expected by the rest of the program. It also post-processes the `calc_vdb` and `calc_mwu` results if the accelerator has set the appropriate flags. It also calls `calc_SegBias` because that function is not implemented in hardware.

4.3.4 Processing Units

Each `glfgen_combine` processing unit (Figure 4.6) consists of two modules: `bcf_call_glfgen` and `bcf_call_combine`. The `bcf_call_glfgen` module reads data from the input FIFO and performs the `bcf_call_glfgen` function and its callees in hardware. The `bcf_call_combine` module reads the results from the `bcf_call_glfgen` module, performs the `bcf_call_combine` function and its callees in hardware, and writes the results to the output FIFO. The two modules can operate in parallel: once `bcf_call_combine` acknowledges that it has latched in the results from `bcf_call_glfgen`, the `bcf_call_glfgen` module resets and processes the next new packet (if any).

4.3.5 Implementation of `bcf_call_glfgen`

The `bcf_call_glfgen` module waits for data to be present in the input FIFO and for an arbitration lock to be granted, then begins reading the data 16 bytes at a time. The first 16 bytes are assumed to be a `glf_header` type, while the rest are assumed to be the type `glf_packed`. If the `valid` bit in the header is not set, then the module asserts `invalid_header`, stops processing, and returns to an idle state until more data arrives. Otherwise, it reads `n` (see the definition of `glf_header` in Listing 4.4) more times from the FIFO. As each base is read, a quality score q is chosen according to Equation 4.1, where $baseQ$ is the adjusted base quality score, $mapQ$ is the mapping quality score, and $seqQ$ is the indel quality score, or 99 if this is not an indel. If $baseQ$ is less than 13, then the read is thrown away.

$$q = \max(4, \min(baseQ, mapQ, seqQ, 63)) \tag{4.1}$$

If the read was not thrown away, the base type is then decoded into 4 bits using either the `base` member or part of the `aux` member if it is an indel. If the base is invalid, then the reference base is used instead. The main loop in `bcf_call_glfgen` keeps track of a number of different statistics (Listing 4.7). The hardware implements all of these operations in parallel. There are 13 counters that track the following:

- The number of bases different from the reference.
- The number of bases matching the reference with a forward read direction.
- The number of bases matching the reference with a reverse read direction.
- The number of bases different from the reference with a forward read direction.
- The number of bases different from the reference with a reverse read direction.

- The number of bases matching the reference at every `epos`.[2]
- The number of bases different from the reference at every `epos`.
- The number of bases matching the reference with a given mapping quality.
- The number of bases different from the reference with a given mapping quality.
- The number of bases matching the reference with a given base quality.
- The number of bases different from the reference with a given base quality.
- The number of bases with a given mapping quality and a forward read direction.
- The number of bases with a given mapping quality and a reverse read direction.

There are six accumulators and six multiply-accumulate units that keep running sums and sums of squares of:

- The base qualities when the bases match the reference.
- The base qualities when the bases are different from the reference.
- The mapping qualities when the bases match the reference.
- The mapping qualities when the bases are different from the reference.
- The minimum distances[3] when the bases match the reference.
- The minimum distances when the bases are different from the reference.

Finally, there is another accumulator that keeps a sum of the observed quality scores for each valid base type. See Figure 4.7 for a block diagram

[2]The position of the base relative to where the read was aligned.
[3]`min_dist`, the minimum distance from either end of the read.

of the accumulators. These counts and sums are used later for calculating various statistics.

Listing 4.7: Statistic counting loop of `bcf_call_glfgen`.

```
for (i = 0; i < _n; ++i) {
  ...
  if (is_diff) ++r->n_supp;
  ...
  if (b < 4) r->qsum[b] += q;
  ++r->anno[0<<2|is_diff<<1|bam_is_rev(p->b)];
  min_dist = p->b->core.l_qseq - 1 - p->qpos;
  if (min_dist > p->qpos) min_dist = p->qpos;
  if (min_dist > CAP_DIST) min_dist = CAP_DIST;
  r->anno[1<<2|is_diff<<1|0] += baseQ;
  r->anno[1<<2|is_diff<<1|1] += baseQ * baseQ;
  r->anno[2<<2|is_diff<<1|0] += mapQ;
  r->anno[2<<2|is_diff<<1|1] += mapQ * mapQ;
  r->anno[3<<2|is_diff<<1|0] += min_dist;
  r->anno[3<<2|is_diff<<1|1] += min_dist * min_dist;

  ...
  if ( bam_seqi(bam_get_seq(p->b),p->qpos) == ref_base )
  {
      bca->ref_pos[epos]++;
      bca->ref_bq[ibq]++;
      bca->ref_mq[imq]++;
  }
  else
  {
      bca->alt_pos[epos]++;
      bca->alt_bq[ibq]++;
      bca->alt_mq[imq]++;
  }
}
```

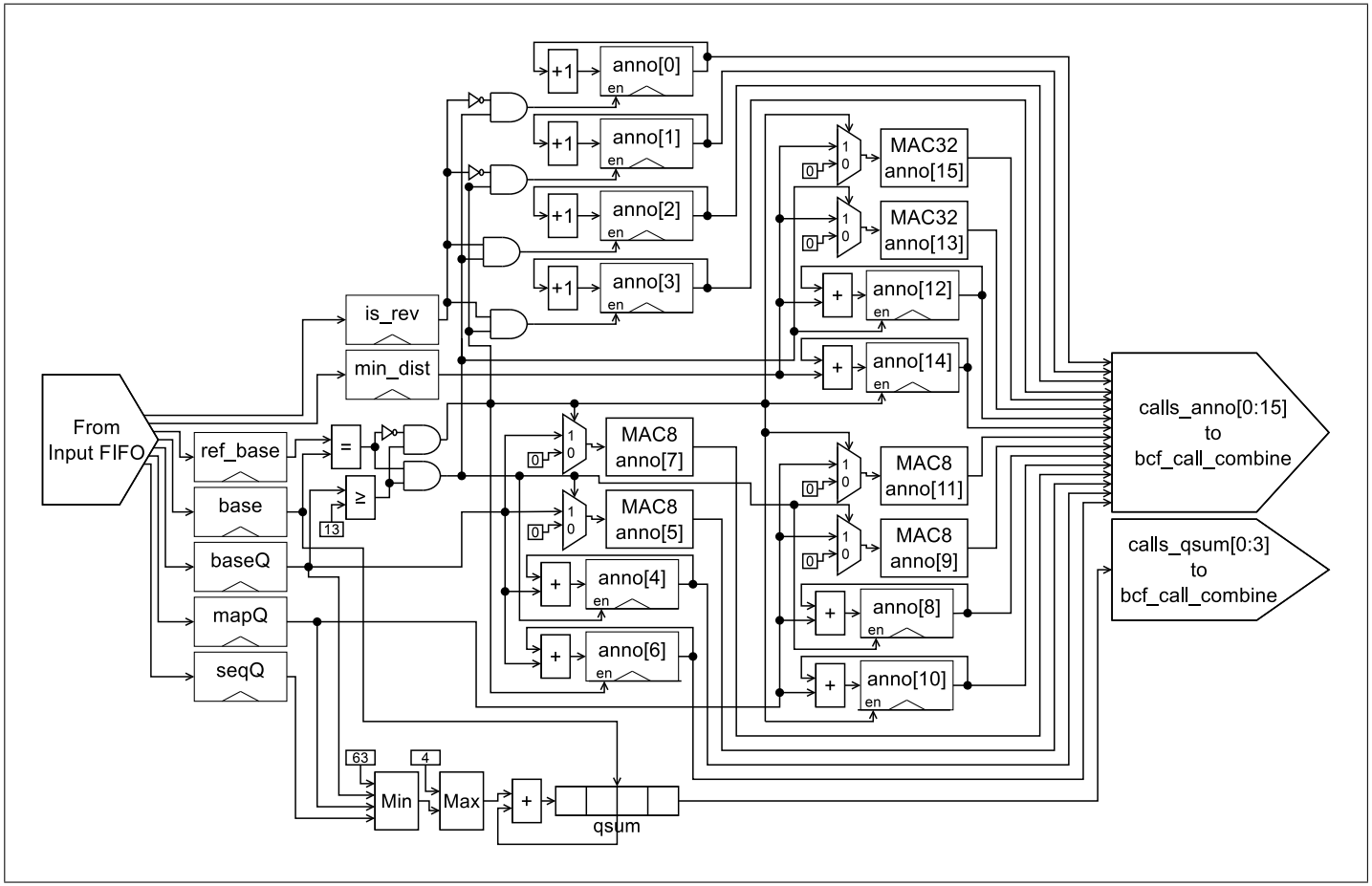

Figure 4.7: Block diagram of the counters and accumulators in `bcf_call_glfgen`. The blocks labeled "MAC8" are multiply-accumulate units that take an input 8 bits wide, square it, and accumulate into a 64 bit value. The blocks labeled "MAC32" take an input 32 bits wide, square it, and accumulate it into a 64 bit value. Taken together, the values returned by these units comprise the `calls_anno` array that is passed to `bcf_call_combine`. The `qsum` array accumulates the quality scores observed for each base type, indexed by `base`. It is passed to `bcf_call_combine` as `calls_qsum`.

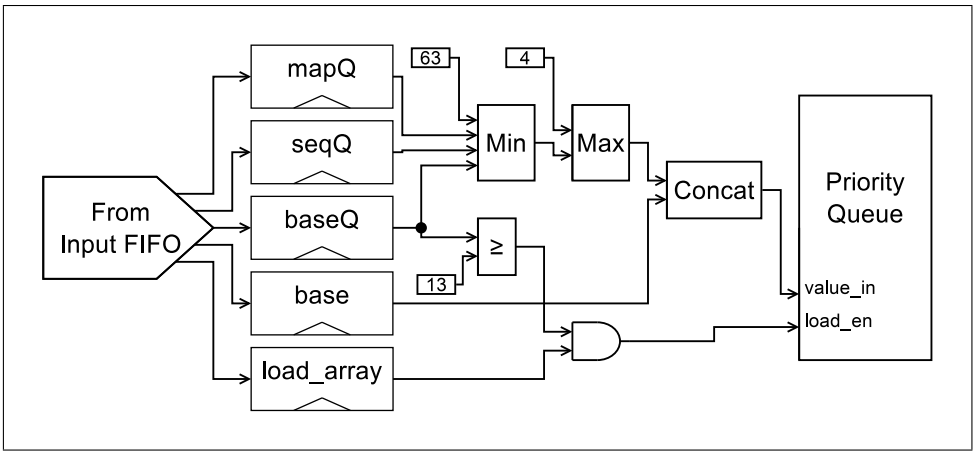

Figure 4.8: Block diagram of how the bases are inserted in the priority queue in `bcf_call_glfgen`. The "Concat" block concatenates the two input values into a single bit vector.

If `load_array` is set to 1, then the base is also inserted into a priority queue along with its quality score in the form $\{q, base\}$ (see Figure 4.8). The queue sorts the bases in descending order for `errmod_cal`. This behavior allows the hardware to mimic how the software implementation of `errmod_cal` takes a random subset of the bases if there are more than 255 available (Listing 4.8). The function `ks_shuffle` randomizes the array and the first 255 entries are used from then on. The hardware will simply take any base with `load_array` set, and it is up to the software implementation to determine which bases to insert.

Listing 4.8: Code that takes a random subset of large data sets.

```
if (n > 255) {
  ks_shuffle(uint16_t, n, bases);
  n = 255;
}
```

When all relevant data has been read from the input FIFO, `bcf_call_glfgen` releases its lock on the FIFO so that another processing unit can begin reading data. When all valid bases have been entered into the priority queue, `bcf_call_glfgen` signals `beta_table` and `errmod_cal` to start. It then waits for `errmod_cal` to finish and signals `bcf_call_combine` that its data is ready. When `bcf_call_combine` acknowledges that the data has been latched, `bcf_call_glfgen` resets so that it can process the next set of reads.

4.3.6 Priority Queue

The priority queue is a shifting structure 13 bits wide and 256 entries deep. When a value is inserted into the queue, it is compared in parallel with all 256 entries. It is then inserted after the largest entry that it is larger than. All entries lower than the inserted value are shifted down. If the queue is full, then the smallest entry will be shifted out of the queue. When data is read back out of the queue, it is returned in descending order. Because the queue feeds a pipelined multiply accumulate operation that does not feature data forwarding, it automatically inserts bubbles into the pipeline as necessary. This feature is discussed in further detail in Section 4.3.7.

4.3.7 Implementation of `errmod_cal`

The `errmod_cal` submodule requires the most time and area of all operations in `bcf_call_glfgen`. In the software version, `errmod_cal` uses precomputed double-precision look-up tables to generate likelihood values. There are three tables, named `lhet`, `fk`, and `beta`. The `lhet` and `fk` tables are sufficiently small (2kB and 512kB, respectively) to be implemented as ROMs. However, the `beta` table is 32MB in size, larger than all the available on-chip memory on the Stratix V. As a result, this table has to be stored in external memory. The DE5 provides 32MB of QDRII+ memory that is used for this purpose. After the driver is loaded, the beta table is copied to the QDRII+ memory. The data will remain in the QDRII+ memory until power is removed from the DE5 board. When `bcf_call_glfgen` has finished loading the priority queue with n valid bases, it triggers the `beta_table` submodule to begin preloading all values from `beta` for n, which requires at most 128kB of on-chip memory. The `beta_table` submodule also acts as a cache, only loading new values if n changes. The caching improves performance in cases where a subsequent call has the same number of valid bases as a previous call.

Once `beta_table` is fully populated, `errmod_cal` begins shifting values out of the priority queue. Running counts are kept of each base type and

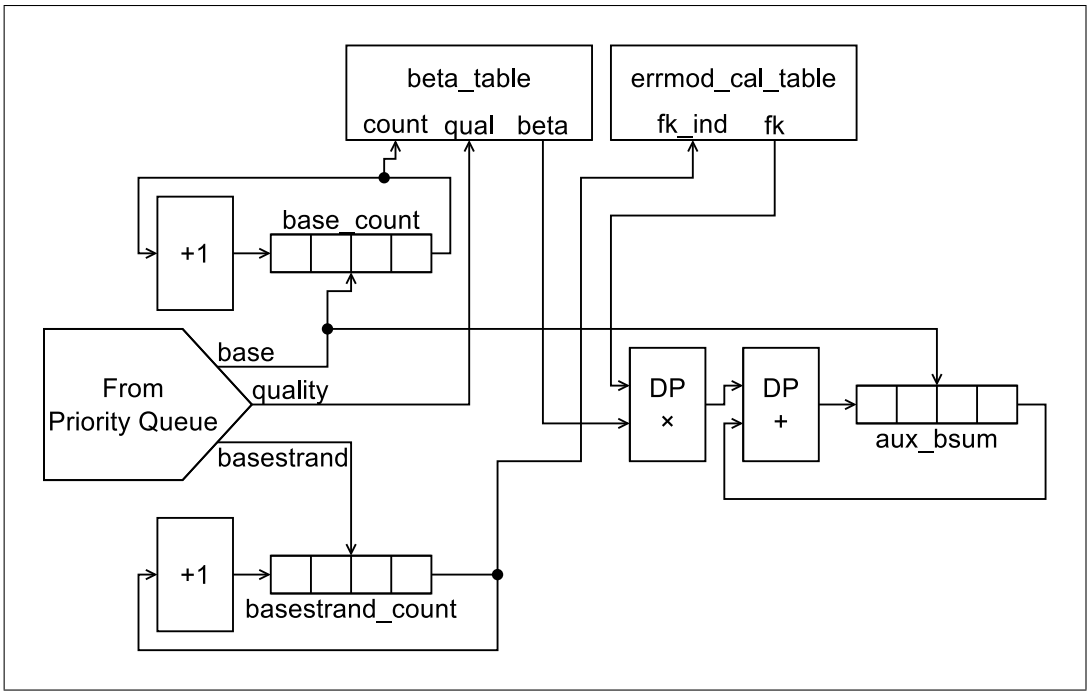

Figure 4.9: Block diagram of how bases are read out of the priority queue by `errmod_cal`, counted, and used to calculate `aux_bsum`. The value of `base` is used to index the `base_count` and `aux_bsum` arrays, while the value of `basestrand` is used to index `basestrand_count`.

each "basestrand" (the combination of the base type and which strand the base came from). The base count and current base quality are used to index `beta_table`, while the basestrand count is used to index `fk`. The `beta` and `fk` values are first multiplied together with a double-precision floating point multiplier, then accumulated with a double-precision adder into an array named `aux_bsum`, indexed by base type. The base type counts are also saved into an array named `aux_c`. This operation corresponds to the loop in Listing 4.9. It should be noted that the `aux.fsum` array on line 10 is only used later to build a sum that is then never referenced again. Therefore, that operation is omitted in the hardware implementation (Figure 4.9).

Listing 4.9: Loop in `errmod_cal` that counts bases and accumulates `fk` and `beta` values.

```
for (j = n - 1; j >= 0; --j) { // calculate esum and fsum
    uint16_t b = bases[j];
    /* extract quality and cap at 63 */
    int qual = b>>5 < 4? 4 : b>>5;
    if (qual > 63) qual = 63;
    /* extract base ORed with strand */
    int basestrand = b&0x1f;
    /* extract base */
    int base = b&0xf;
    aux.fsum[base] += em->coef->fk[w[basestrand]];
    aux.bsum[base] += em->coef->fk[w[basestrand]]
                    * em->coef->beta[qual<<16|n<<8|aux.c[base]];
    ++aux.c[base];
    ++w[basestrand];
}
```

Because the multiplier and adder are pipelined (the multiplier with a depth of 5 stages, the adder with a depth of 7 stages), it is possible to perform several of these computations in parallel. However, there are data dependency issues that arise if there are two entries for the same base type in the adder at the same time, since there is no data forwarding mechanism provided by the adder IP. Using separate double-precision accumulators would alleviate this problem, but would incur a very high area cost. Instead, the priority queue has a state variable that cycles through the base types (0, 1, 2, 3, 4) in order. If the base at the head of the queue matches the currently selected base type, then the base is shifted out to `errmod_cal`. Otherwise, a value with an invalid base type (5) is shifted out, which acts as a bubble in the adder pipeline. If the queue cycles through all of the base types, then 3 more bubbles are inserted. These bubbles ensure that any dependent data will have left the pipeline when the base type select returns to 0.

Once all of the bases have been shifted out of the priority queue and counted, the `errmod_cal` module waits for the multiply accumulate pipeline to clear. When the pipeline is clear, `errmod_cal` begins loading `lhet` values into registers. The `lhet` values are used to calculate the likelihoods for the

heterozygous[4] case. For two heterozygous bases with types i and j, $j > i$, the `lhet` ROM is indexed by the address $\{aux\_c[i] + aux\_c[j], aux\_c[j]\}$. There are 10 combinations of base types under these criteria, so 10 values must be fetched from `lhet`.

In parallel with the `lhet` fetches, the `errmod_cal` module also begins calculating the `tmp1` values in Listing 4.10. The first for loop (the homozygous case) generates its `tmp1` value by summing every combination of the `aux_bsum` values for 4 distinct base types.[5] The second for loop (the heterozygous case) generates its `tmp1` value by summing every combination of the `aux_bsum` values for 3 distinct base types.[6] Some of the results from the heterozygous `tmp1` calculations can be used to generate the homozygous `tmp1` values, so the heterozygous `tmp1` values are calculated first, using 3 double precision floating point adders operating in parallel. The addition operations are all ordered to maximize utilization of the addition pipelines. Once the heterozygous `tmp1` values are calculated, they are then used to calculate the final values for the homozygous and heterozygous cases by adding either the remaining `aux_bsum` value or the respective `lhet` value. Since the `tmp2` values for both cases are checked only for whether they are 0 or not, the final values are chosen based on a logical-or reduction of the respective `aux_c` values rather than their sum.

[4]Heterozygous bases occur when a cell has two different versions of a gene in its chromosomes.

[5]$\{0, 1, 2, 3\}, \{0, 1, 2, 4\}, ..., \{1, 2, 3, 4\}$

[6]$\{0, 1, 2\}, \{0, 1, 3\}, ..., \{2, 3, 4\}$

Listing 4.10: Source of the likelihood calculation loop in `errmod_cal`.

```
for (j = 0; j < m; ++j) {
  float tmp1, tmp3;
  int tmp2;
  // Homozygous case
  for (k = 0, tmp1 = tmp3 = 0.0, tmp2 = 0; k < m; ++k) {
    if (k == j) continue;
    tmp1 += aux.bsum[k];
    tmp2 += aux.c[k];
    tmp3 += aux.fsum[k];
  }
  if (tmp2) {
    q[j*m+j] = tmp1;
  }
  // Heterozygous case
  for (k = j + 1; k < m; ++k) {
    int cjk = aux.c[j] + aux.c[k];
    for (i = 0, tmp2 = 0, tmp1 = tmp3 = 0.0; i < m; ++i) {
      if (i == j || i == k) continue;
      tmp1 += aux.bsum[i];
      tmp2 += aux.c[i];
      tmp3 += aux.fsum[i];
    }
    if (tmp2) {
      q[k*m+j] = -4.343 * em->coef->lhet[cjk<<8|aux.c[k]]
                + tmp1;
      q[j*m+k] = q[k*m+j];
    } else {
      q[k*m+j] = -4.343 * em->coef->lhet[cjk<<8|aux.c[k]];
      q[j*m+k] = q[k*m+j];
    }
  }
  for (k = 0; k < m; ++k) {
    if (q[j*m+k] < 0.0) {
      q[j*m+k] = 0.0;
    }
  }
}
```

4.3.8 Implementation of `bcf_call_combine`

The `bcf_call_combine` module reads the results from `bcf_call_glfgen`, writes data to the output FIFO, performs some calculations, and coordinates several submodules. The FIFO operations are handled in a separate state machine from the computation state machine. A high level block diagram of the `bcf_call_combine` module is shown in Figure 4.10.

The FIFO state machine writes the results of computation into the output FIFO, formatted as a `pcie_packet_in` type. It first waits for the calculation of `call_pl` to complete, then writes those results to the FIFO. Then, it waits for the `write_qsum` signal, at which point it writes the `qsum`, `n`, `n_alleles`, `shift`, and `ori_ref` members. Then, it waits for the `call_anno_done` and `call_shift_done` signals, and writes `unseen`, `depth`, `ori_depth`, `mq0`, `bca_anno`, and `anno`. Finally, it waits for the `start_write` signal and writes the rest of the members.

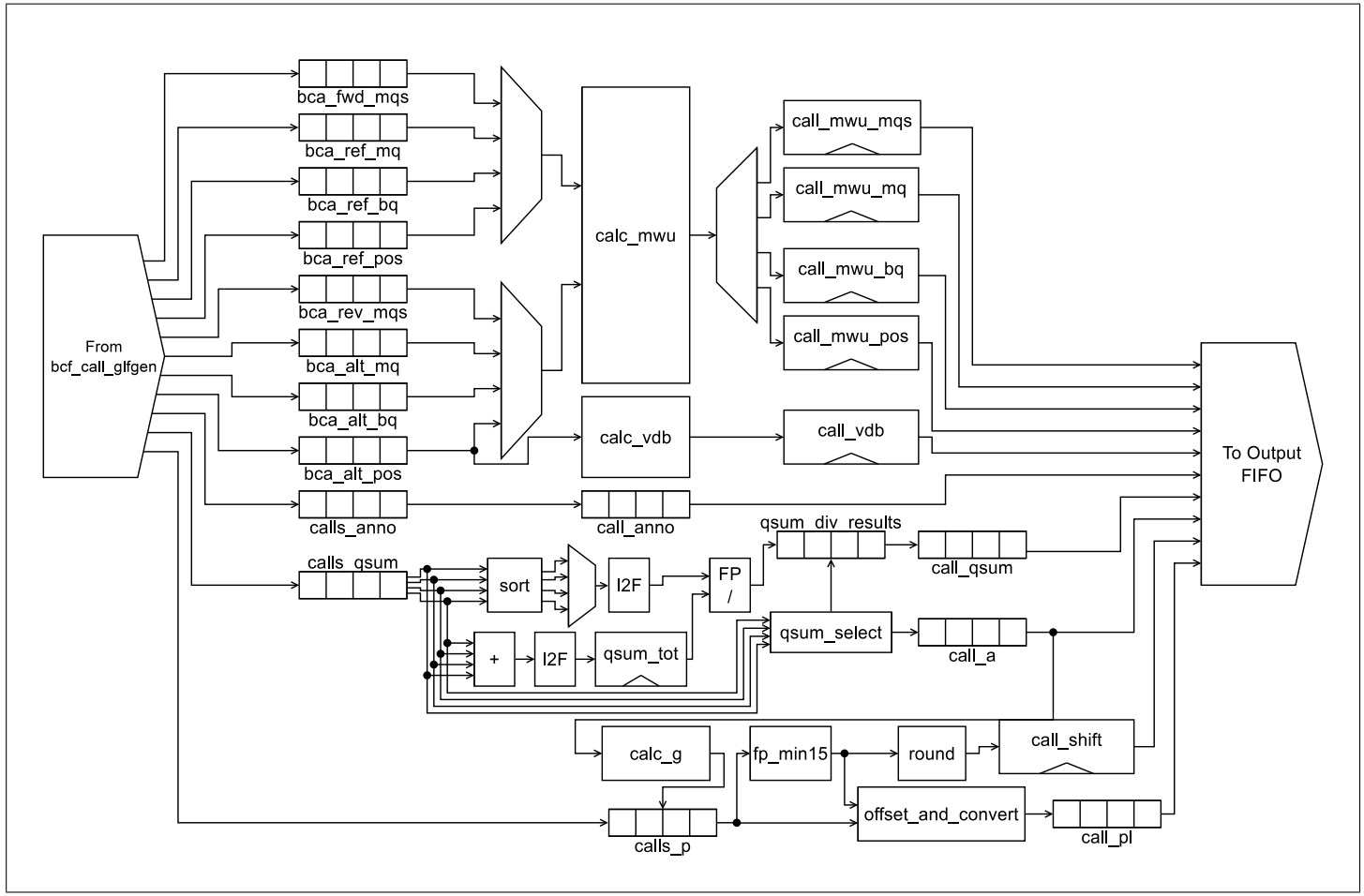

Figure 4.10: Block diagram of the `bcf_call_combine` hardware implementation. Blocks marked "I2F" are integer to single-precision conversion blocks. The `calc_mwu` block selects two arrays at a time from the 8 `bca` arrays, one `ref` (or `fwd`) and one `alt` (or `rev`). It then reads 4 words at a time from those arrays. `calc_vdb` reads 4 words at a time from the `bca_alt_pos` array. `calls_anno` is passed through as `call_anno`. The 4 element array `calls_qsum` is summed with a 4-way parallel adder, then each value is converted to floating point and normalized by the sum. These values are then indexed by `qsum_select` into `call_qsum`. `qsum_select` also generates the `call_a` values, which are then used to generate the `g` values that index `calls_p`.

The computation state machine implements the body of the `bcf_call_combine` function. When the computation state machine receives the `start` signal from the `bcf_call_glfgen` module, it does the following:

1. Latch all output values of `bcf_call_glfgen` into local registers so that `bcf_call_glfgen` can begin work on any new data.

2. Sum the quality score sums for each base (`qsum`) to get the total quality score sum (`qsum_tot`). This operation is done in one cycle with a 4-way parallel adder built from Altera's provided IP library. The software implementation is shown in Listing 4.11.

Listing 4.11: Summation of `qsum` in `bcf_call_combine`.

```
int qsum_tot=0;
for (j=0; j<4; j++) {
  qsum_tot += qsum[j]; call->qsum[j] = 0;
}
```

3. Encode each `qsum` with its respective base in the lowest two bits (Listing 4.12), ensuring that the bases and quality sums are always associated in the operations that follow.

Listing 4.12: Encoding the base in the `qsum` values.

```
for (j=0; j<4; j++) {
    qsum[j] = qsum[j] << 2 | j;
}
```

4. Set the `depth`, `ori_depth`, `mq0`, and `anno` fields of the `call` structure to their respective values in the `calls` structure. Listing 4.13 shows the general case for any number of samples.

Listing 4.13: Setting the `depth`, `ori_depth`, `mq0`, and `anno` fields.

```
memset(call->anno, 0, 16 * sizeof(double));
call->ori_depth = 0;
call->depth     = 0;
call->mq0       = 0;
for (i = 0; i < n; ++i) {
  call->depth += calls[i].depth;
  call->ori_depth += calls[i].ori_depth;
  call->mq0 += calls[i].mq0;
  for (j = 0; j < 16; ++j)
    call->anno[j] += calls[i].anno[j];
}
```

5. Sort the encoded `qsum` values in ascending order (Listing 4.14), which effectively sorts the possible alleles by their quality score sums. This sort is accomplished in 3 cycles with an optimal 4-way sorting network.

Listing 4.14: Sorting the `qsum` values.

```
  for (j = i; j > 0 && qsum[j] < qsum[j-1]; --j)
    tmp = qsum[j], qsum[j] = qsum[j-1], qsum[j-1] = tmp;
```

6. Divide each of the sorted `qsum` values by `qsum_tot` as in Listing 4.15. These quotients are in the range $[0, 1]$, where a value closer to 1 indicates a higher quality base. This divide is accomplished in 25 cycles with a single pipelined floating point divider built from Altera's IP.

Listing 4.15: Assignment loop for `call->qsum` and `call->a`.

```
call->a[0] = ref4;
for (i = 3, j = 1; i >= 0; --i)
{
  if ((qsum[i]&3) == ref4) {
    call->qsum[0] =
      qsum_tot ? (float)(qsum[i]>>2)/qsum_tot : 0;
  }
  else {
    if ( !(qsum[i]>>2) ) break;
    call->qsum[j] = (float)(qsum[i]>>2)/qsum_tot;
    call->a[j++]  = qsum[i]&3;
  }
}
```

7. The next step implements the `for` loop in Listing 4.15. This loop assigns the quotients from step 6 to `call->qsum` in the order $\{REF$, ALT_1, ALT_2, ..., $ALT_3\}$, where REF is the reference base, and ALT_i is the i-th highest quality observed non-reference base. The bases themselves are assigned to `call->a` in the same order. The value of `j` at the end of the loop is therefore the number of different bases observed in the aligned reads. This operation is implemented in hardware by unrolling the `for` loop into all possible combinations of the comparisons in lines 4 and 8. Unrolling this loop allows the assignments to be accomplished in a single cycle, after which the `write_qsum` signal is raised.

8. If we are looking for SNPs, it adds the first "unseen" base (the first base with a quality score sum of 0) to the end of the `call->a` array and points `call->unseen` to the position of that base. If we are looking for indels and there is only one observed allele, then the state machine sets the return value to -1 and raises the `start_write` signal.

9. This step implements the loop in Listing 4.16. The multiply accumulate operation was replaced by a small lookup table named `g_lut`, since the members of `call->a` can only take the values $-1, 0, 1, 2, 3, 4$. However, any members of `call->a` that are negative have an index higher than `call->n_alleles`, so that case can be ignored. As a result, there are only $5 \times 5 = 25$ possible values over all combinations of two values of `call->a`. Since `call->n_alleles` ranges between $[1, 5]$, at most 15 entries of `g` will be modified. 15 instances of `g_lut` are able to generate all of the entries in 1 cycle.

Listing 4.16: Assignment loop for `g`.

```
for (i = z = 0; i < call->n_alleles; ++i) {
  for (j = 0; j <= i; ++j) {
    g[z++] = call->a[j] * 5 + call->a[i];
  }
}
```

10. The length of the Phred likelihood array (`x` in Listing 4.17) is calculated. This array corresponds to an upper or lower triangle of the

NxN matrix of possible diploid genotypes (where N equals the number of alleles after step 8). This value is the N-th triangular number, calculated as $\frac{N(N+1)}{2}$. Since the value of N is confined to the range $[1, 5]$, this calculation is implemented as a lookup table.

Listing 4.17: Calculation of `x`.

```
x = call->n_alleles * (call->n_alleles + 1) / 2;
```

11. This step performs the first half of the loop in Listing 4.18, which finds the minimum Phred likelihood from the array calculated in `bcf_call_glfgen` (See Section 3.2). This operation is implemented in hardware with a module named `fp_min_15`. This module consists of a 15-way network that takes advantage of an Altera IP block for a floating point $\min(n)$ function. In addition, it accounts for the fact that `x` (calculated in step 10) only takes on the values $\{1, 3, 6, 10, 15\}$ and simplifies the network if possible. As a result, it is able to find the minimum value in 1 to 4 cycles, depending on the value of `x`.

Listing 4.18: Final Phred likelihood calculation loop.

```
for (i = 0; i < n; ++i)
{
  uint32_t *PL = call->PL + x * i;
  const bcf_callret1_t *r = calls + i;
  float min = FLT_MAX;
  for (j = 0; j < x; ++j) {
    if (min > r->p[g[j]]) min = r->p[g[j]];
  }
  sum_min += min;
  for (j = 0; j < x; ++j) {
    int y;
    y = (int)(r->p[g[j]] - min + .499);
    if (y > 255) y = 255;
    PL[j] = y;
  }
}
```

12. The second half of the loop in Listing 4.18 is performed in the next step, which translates each Phred likelihood score by the minimum

value and then performs an approximate round half down,[7] saturating at 255. This operation is performed in 15 parallel instances of a module named `fp_offset_and_convert`, which uses an Altera IP floating point adder/subtractor (selectable by an input) to perform a floating point subtract between `r->p[g[j]]` and the minimum value found in step 12. The result of the subtraction is then added to the floating point constant 0.499 and the sum is converted to an integer, limited to 255. Only the first `x` entries are written, again taking advantage of the fact that `x` only takes a few discrete values. When all of the `PL` values are written, a signal is raised so that the FIFO state machine can begin writing them to the output FIFO. Because there is only one sample, the value of `sum_min` is identical to `min`.

13. At this point, the state machine waits for all of the FIFO writes to complete, then returns to the `IDLE` state when the `start` signal is lowered.

4.3.9 Implementation of `calc_mwu_bias`

This submodule (Figure 4.11) implements the `calc_mwu_bias` function, which requires the most time of any of the callees of `bcf_call_combine`. One of the first optimizations made in the hardware implementation is the complete elimination of the calculation of `ties`, since later lines that reference it are commented out and the value is never used.

[7] For a floating point value $n \in [i, i+1)$, its rounded value n' will be

$$n' = \begin{cases} i & \text{if } i \leq n \leq i + 0.5 \\ i+1 & \text{otherwise} \end{cases}$$

Listing 4.19: Calculation of `na`, `nb`, and `U`.

```
int na = 0, nb = 0, i;
double U = 0, ties = 0;
for (i=0; i<n; i++)
{
    na += a[i];
    U  += a[i] * (nb + b[i]*0.5);
    nb += b[i];
    if ( a[i] && b[i] )
    {
        double tie = a[i] + b[i];
        ties += (tie*tie-1)*tie;
    }
}
...
// Correction for ties:
//      double N = na+nb;
//      double var2 = (N*N-1)*N-ties;
//      if ( var2==0 ) return 1.0;
//      var2 *= ((double)na*nb)/N/(N-1)/12.0;
// No correction for ties:
...
```

The second optimization was converting the calculation of `U` and `mean` to 1.52.1 format fixed point, since they will always either be integers or end in .5. The `for` loop that calculates `na`, `nb`, and `U` is also unrolled 4 times to parallelize the calculation. For `na` and `nb`, the unrolling is trivial and the calculation can be accomplished with two 4-way parallel adders from Altera's IP blocks.

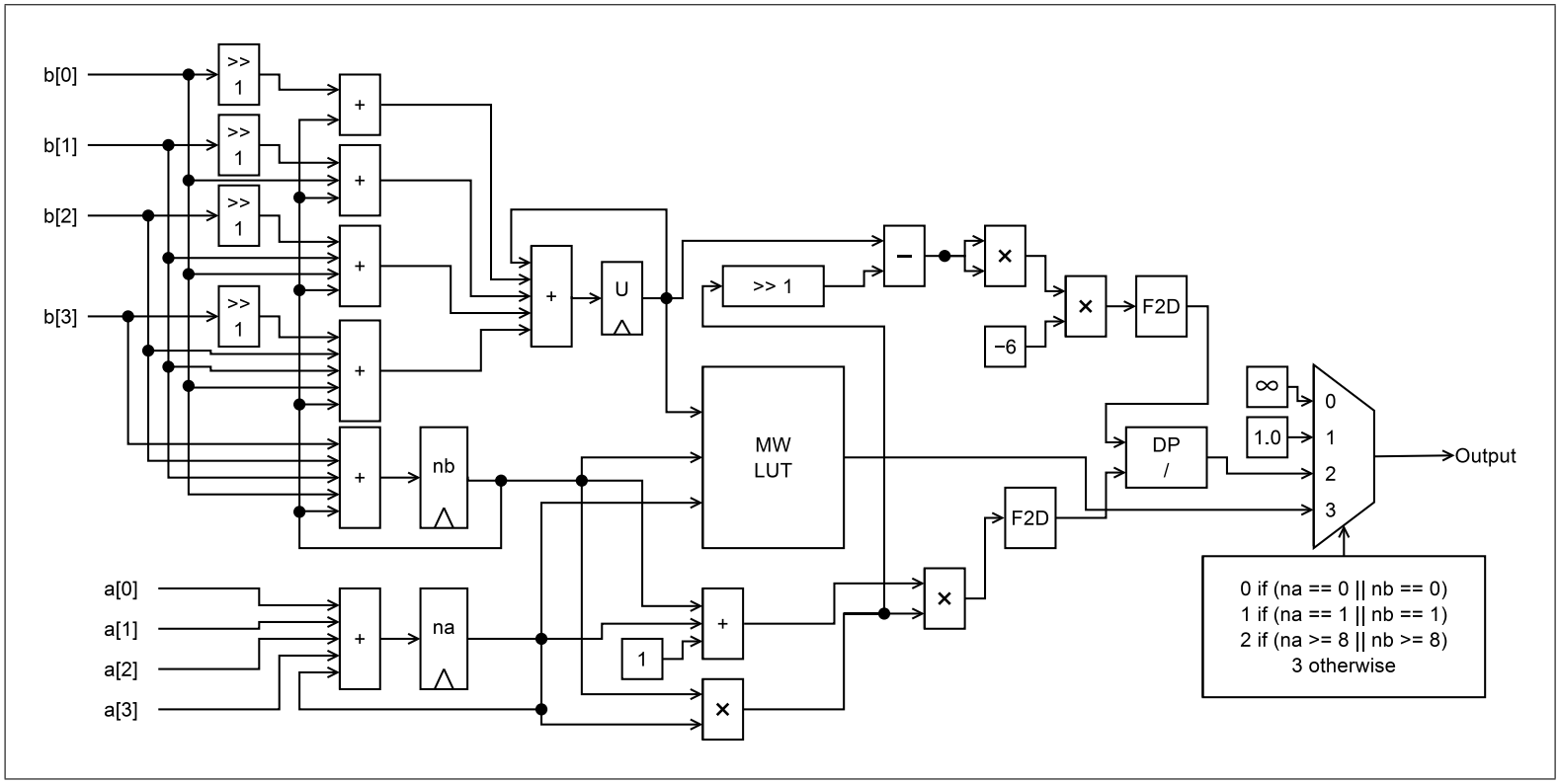

Figure 4.11: Block diagram of the `calc_mwu` hardware implementation. F2D blocks convert from 1.52.1 fixed point to double-precision floating point.

Unrolling `U` results in Listing 4.20.

Listing 4.20: Unrolled calculation of `U`.

```
for (i=0; i<n; i+=4)
{
  U  +=  a[i] * (nb + b[i]*0.5)
      + a[i+1] * (nb + b[i] + b[i+1]*0.5)
      + a[i+2] * (nb + b[i] + b[i+1] + b[i+2]*0.5)
      + a[i+3] * (nb + b[i] + b[i+1] + b[i+2] + b[i+3]*0.5);
}
```

The innermost multiplication by 0.5 can be achieved with a simple shift since fixed point is used. So, the inner sums are computed with parallel fixed point adders of widths 2, 3, 4, and 5. The multiplications are then performed with 4 fixed point multipliers, then the final addition is performed with another 5-way parallel adder.

The calculation of `var2` and the normal approximation result were altered from Listing 4.21 to Listing 4.22:

Listing 4.21: Original calculation of `var2`.

```
double var2 = ((double)na*nb)*(na+nb+1)/12.0;
if ( na>=8 || nb>=8 )
{
    // Normal approximation, very good for na>=8 && nb>=8
    // and reasonable if na<8 or nb<8
    return exp(-0.5*(U-mean)*(U-mean)/var2);
}
```

Listing 4.22: Changes to calculation of `var2` and normal approximation.

```
double var2 = ((double)na*nb)*(na+nb+1);
if ( na>=8 || nb>=8 )
{
    // Normal approximation, very good for na>=8 && nb>=8
    // and reasonable if na<8 or nb<8
    return exp(-6.0*(U-mean)*(U-mean)/var2);
}
```

This change allows `var2` to be calculated as an integer and the numerator of the normal approximation to be calculated in fixed point. Floating point arithmetic is not required until the final division and exponential operations. In order to reduce hardware usage, the exponential was left in software as a post-processing step based on a status flag. All floating point calculations had to be left as double-precision to avoid overflows at locations with large alignment depths.

Listing 4.23: Source of `mann_whitney_1947`.

```
double mann_whitney_1947(int n, int m, int U)
{
    if (U<0) return 0;
    if (n==0||m==0) return U==0 ? 1 : 0;
    return (double)n/(n+m)*mann_whitney_1947(n-1,m,U-m) +
      (double)m/(n+m)*mann_whitney_1947(n,m-1,U);
}
```

The most significant optimization was made by noting that, as Listing 4.23 shows, the function `mann_whitney_1947` is doubly recursive. Directly implementing the function in hardware was deemed too difficult, not to mention computationally intensive due to the amount of floating point operations. Luckily, it is possible to precalculate these values and store them in a ROM.

First of all, the fact that `mann_whitney_1947` is only called if $2 < \texttt{na} < 8$ and $2 < \texttt{nb} < 8$ (and that `na` and `nb` are both integers) means that the number of possible values is finite. However, the exact value of `U` is not easy to determine without knowing the exact values of the arrays `a` and `b`.

Listing 4.24: Calls to `calc_mwu_bias`.

```
call->mwu_pos =
  calc_mwu_bias(bca->ref_pos, bca->alt_pos, bca->npos);
call->mwu_mq  =
  calc_mwu_bias(bca->ref_mq,  bca->alt_mq,  bca->nqual);
call->mwu_bq  =
  calc_mwu_bias(bca->ref_bq,  bca->alt_bq,  bca->nqual);
call->mwu_mqs =
  calc_mwu_bias(bca->fwd_mqs, bca->rev_mqs, bca->nqual);
```

Listing 4.25: Allocation of arrays passed to `calc_mwu_bias`.

```
bca->npos = 100;
bca->ref_pos = malloc(bca->npos*sizeof(int));
bca->alt_pos = malloc(bca->npos*sizeof(int));
bca->nqual = 60;
bca->ref_mq  = malloc(bca->nqual*sizeof(int));
bca->alt_mq  = malloc(bca->nqual*sizeof(int));
bca->ref_bq  = malloc(bca->nqual*sizeof(int));
bca->alt_bq  = malloc(bca->nqual*sizeof(int));
bca->fwd_mqs = malloc(bca->nqual*sizeof(int));
bca->rev_mqs = malloc(bca->nqual*sizeof(int));
```

If we inspect where `calc_mwu_bias` is called (Listing 4.24), we see that its {`a`, `b`} arguments are {`bca->ref_pos`, `bca->alt_pos`}, {`bca->ref_mq`, `bca->alt_mq`}, {`bca->ref_bq`, `bca->alt_bq`}, or {`bca->fwd_mqs`, `bca->rev_mqs`}. As Listing 4.25 shows, these arrays are either 100 or 60 elements long, which makes enumerating all possible permutations infeasible. But, there is an easy to calculate upper bound for `U`, which was determined as follows. From the `for` loop at line 5 of Listing 4.3.9, we get that

$$n_a = \sum_{i=0}^{n-1} a_i \tag{4.2}$$

$$n_b = \sum_{i=0}^{n-1} b_i \tag{4.3}$$

$$U = \sum_{i=0}^{n-1} a_i \left(\sum_{j=0}^{i-1} b_j + b_i \times 0.5 \right)$$

$$U = \sum_{i=0}^{n-1} a_i \left(\sum_{j=0}^{i-1} b_j + b_i - b_i \times 0.5 \right)$$

$$U = \sum_{i=0}^{n-1} a_i \left(\sum_{j=0}^{i} b_j - b_i \times 0.5 \right)$$

$$U = \sum_{i=0}^{n-1} \left(a_i \sum_{j=0}^{i} b_j - a_i b_i \times 0.5 \right)$$

$$U = \sum_{i=0}^{n-1} a_i \sum_{j=0}^{i} b_j - 0.5 \sum_{i=0}^{n-1} a_i b_i \tag{4.4}$$

If we multiply Equations 4.2 and 4.3, we get

$$n_a \times n_b = \left(\sum_{i=0}^{n-1} a_i\right)\left(\sum_{i=0}^{n-1} b_i\right)$$
$$n_a \times n_b = \sum_{i=0}^{n-1} a_i \left(\sum_{i=0}^{n-1} b_i\right) \tag{4.5}$$

Listing 4.26: Initialization of arrays passed to `calc_mwu_bias`.

```
memset(bca->ref_pos,0,sizeof(int)*bca->npos);
memset(bca->alt_pos,0,sizeof(int)*bca->npos);
memset(bca->ref_mq,0,sizeof(int)*bca->nqual);
memset(bca->alt_mq,0,sizeof(int)*bca->nqual);
memset(bca->ref_bq,0,sizeof(int)*bca->nqual);
memset(bca->alt_bq,0,sizeof(int)*bca->nqual);
memset(bca->fwd_mqs,0,sizeof(int)*bca->nqual);
memset(bca->rev_mqs,0,sizeof(int)*bca->nqual);
```

Listing 4.27: Setting values of arrays passed to `calc_mwu_bias`.

```
if ( bam_seqi(bam_get_seq(p->b),p->qpos) == ref_base )
{
  bca->ref_pos[epos]++;
  bca->ref_bq[ibq]++;
  bca->ref_mq[imq]++;
}
else
{
  bca->alt_pos[epos]++;
  bca->alt_bq[ibq]++;
  bca->alt_mq[imq]++;
}
```

From Listings 4.26 and 4.27, we see that all elements of those arrays are greater than or equal to 0. Therefore,

$$\sum_{j=0}^{m} b_j \leq \sum_{i=0}^{n-1} b_i \text{ if } m \leq n-1$$
$$-0.5 \sum_{i=0}^{n-1} a_i b_i \leq 0$$

So, we can then determine a relation between Equations 4.4 and 4.5:

$$\sum_{i=0}^{n-1} a_i \sum_{j=0}^{i} b_j - 0.5 \sum_{i=0}^{n-1} a_i b_i \leq \sum_{i=0}^{n-1} a_i \left(\sum_{i=0}^{n-1} b_i \right)$$
$$U \leq n_a n_b \tag{4.6}$$

Using this relation, we can precalculate all values of `calc_mwu_bias` for the aforementioned range of `na` and `nb`. To further speed up calculation, the range was expanded to $2 \leq$ `na` < 8 and $2 \leq$ `nb` < 8. This change includes the linear approximation case for small values, speeding up those calculations. Finally, although `U` is calculated as a floating point value, it is truncated to an integer during the call to `mann_whitney_1947`. As a result, it is possible to precalculate all possible values of `mann_whitney_1947` for $0 \leq$ `na` ≤ 7, $0 \leq$ `nb` ≤ 7, and $0 \leq$ `U` ≤ 49, and store them in a ROM with a 12-bit address in the form `(na « 9) | (nb « 6) | (U & 0x3F)`. It is important to note that `U` can also be negative, but `mann_whitney_1947` always returns 0 when `U` < 0, so this case is easily handled outside of the ROM.

The linear approximation case (Listing 4.28) still needs to be calculated if one of `na` or `nb` equals 2 and the other is greater than 8. This case is fairly straightforward. If `U` $>$ `mean`, then the numerator is calculated in fixed point with a shift and a subtraction. The result (or `U` if `U` *le* `mean`) is then converted to double-precision floating point along with `mean`, and the result is calculated with a floating point divide.

Listing 4.28: Linear approximation for `calc_mwu_bias`.

```
if ( na==2 || nb==2 )
{
  return U>mean ? (2.0*mean-U)/mean : U/mean;
}
```

4.3.10 Implementation of `calc_SegBias`

The computations in `calc_SegBias` are highly dependent on double precision floating point in order to avoid overflow. In addition, it is responsible for only 0.20% of total run time. Therefore, the execution of this function was left in software to reduce hardware usage.

4.3.11 Implementation of `calc_vdb`

This submodule implements the Variant Distance Bias calculation, and allows for some optimization as well. First of all, the first for loop (Listing 4.29) that calculates `dp` and the numerator of `mean_pos` is unrolled 4 times, allowing the additions to be performed by 4-way parallel adders. Also, the final division by `dp` is omitted.

Listing 4.29: Calculation of `dp` and `mean_pos`.

```
int i, dp = 0;
float mean_pos = 0, mean_diff = 0;
for (i=0; i<npos; i++)
{
  if ( !pos[i] ) continue;
  dp += pos[i];
  mean_pos += pos[i]*i;
}
if ( dp<2 ) return HUGE_VAL;

mean_pos /= dp;
```

This change transforms the definition of `mean_pos` to `mean_pos`$'$, as seen in Equation 4.7. Figure 4.12 shows how this step is implemented in hardware.

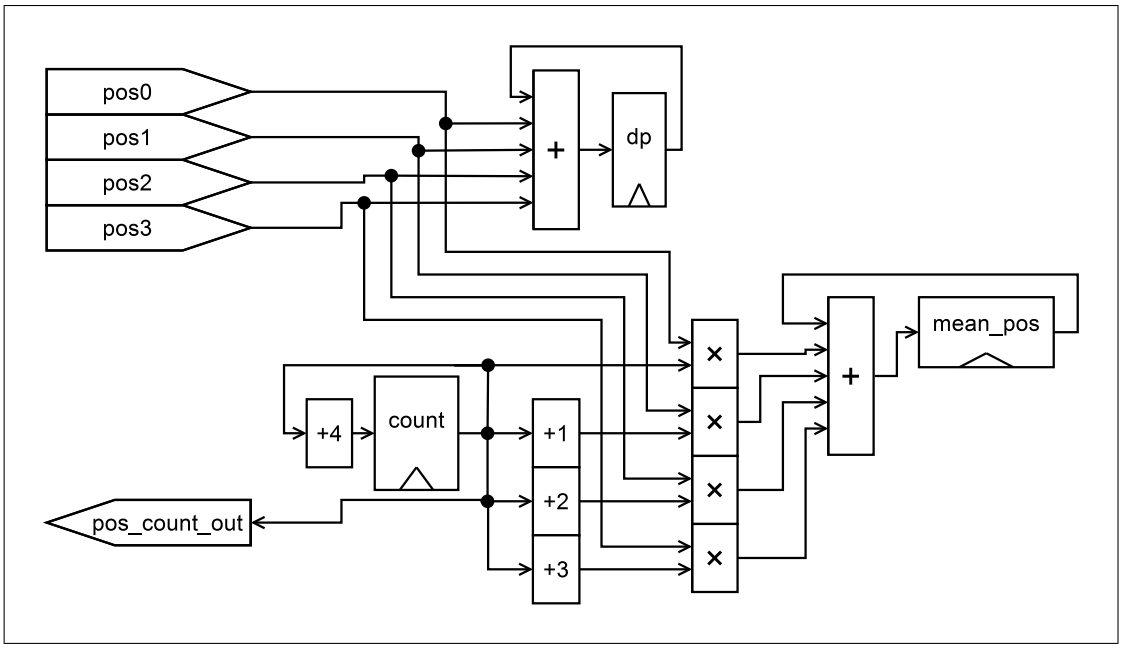

Figure 4.12: Block diagram of the `dp` and `mean_pos` calculation in the hardware implementation of `calc_vdb`. `pos_count_out` is used to index the `bca_alt_pos` array in `bcf_call_combine`.

Equations 4.9 and 4.10 show how `mean_diff` (normally calculated as in Listing 4.30) can be calculated using $\texttt{mean\_pos}'$.

$$\texttt{mean\_pos}' = dp \times \texttt{mean\_pos} = \sum_{0}^{99} pos_i \times i \tag{4.7}$$

$$\texttt{mean\_diff} = \frac{\sum_0^{99} pos_i \, |i - \texttt{mean\_pos}|}{dp} \tag{4.8}$$

$$\texttt{mean\_diff} = \frac{|dp| \sum_0^{99} pos_i \, |i - \texttt{mean\_pos}|}{|dp| \, dp} \tag{4.9}$$

$$\texttt{mean\_diff} = \frac{\sum_0^{99} pos_i \, |dp \times i - dp \times \texttt{mean\_pos}|}{|dp| \, dp}$$

$$\texttt{mean\_diff} = \frac{\sum_0^{99} pos_i \, |dp \times i - \texttt{mean\_pos}'|}{|dp| \, dp} \tag{4.10}$$

Listing 4.30: Calculation of `mean_diff`.

```
for (i=0; i<npos; i++) {
  if ( !pos[i] ) continue;
  mean_diff += pos[i] * fabs(i - mean_pos);
}
```

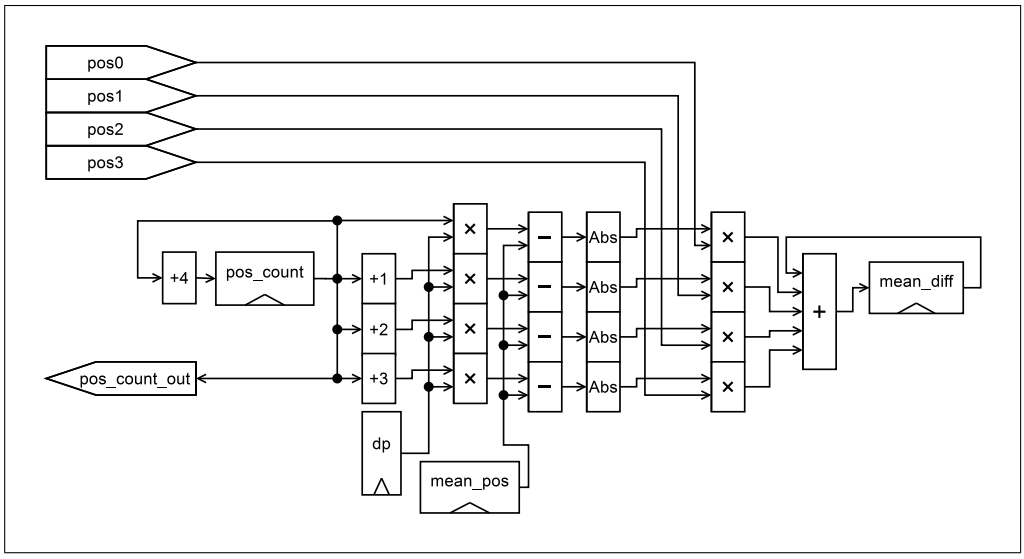

Figure 4.13: Block diagram of the `mean_diff` calculation in the hardware implementation of `calc_vdb`. `pos_count_out` is used to index the `bca_alt_pos` array in `bcf_call_combine`. "Abs" blocks find the absolute value of their input by taking the 2's complement if the input is negative.

This change allows us to calculate the for loop with only integer operations, then convert to floating point for the division by $|dp|\,dp$. Figure 4.13 shows a block diagram of how this calculation is implemented in hardware.

Listing 4.31: Calculation of the return value in the `dp` = 2 case.

```
if ( dp==2 )
{
  return (2*readlen-2*(ipos+1)-1)*(ipos+1)/
    (readlen-1)/(readlen*0.5);
}
```

The return value in Listing 4.31 can also be optimized. Since `readlen` is defined to be 100, we get:

$$
\begin{aligned}
ret &= (2\,\texttt{readlen} - 2(\texttt{ipos}+1) - 1)(\texttt{ipos}+1)/(\texttt{readlen}-1)/(\texttt{readlen}\times 0.5)\\
ret &= (200 - 2(\texttt{ipos}+1) - 1)(\texttt{ipos}+1)/99/50.0\\
ret &= (199 - 2\,\texttt{ipos} - 2)(\texttt{ipos}+1)/99/50.0\\
ret &= (197 - 2\,\texttt{ipos})(\texttt{ipos}+1)/99/50.0\\
ret &= (197\,\texttt{ipos} - 2\,\texttt{ipos}^2 + 197 - 2\,\texttt{ipos})/99/50.0\\
ret &= (195\,\texttt{ipos} - 2\,\texttt{ipos}^2 + 197)/99/50.0
\end{aligned}
\tag{4.11}
$$

The numerator of Equation 4.11 can be calculated solely with shifts and additions (with the exception of $\texttt{ipos}^2$, which is calculated with an integer multiply Altera IP block).

$$
\begin{aligned}
195\,\texttt{ipos} &= (128+64+2+1)\texttt{ipos}\\
195\,\texttt{ipos} &= (\texttt{ipos} << 7) + (\texttt{ipos} << 6) + (\texttt{ipos} << 1) + \texttt{ipos}\\
2\,\texttt{ipos}^2 &= (\texttt{ipos}^2 << 1)\\
ret &= ((\texttt{ipos} << 7) + (\texttt{ipos} << 6) + (\texttt{ipos} << 1) + \texttt{ipos} + 197\\
&\quad - (\texttt{ipos}^2 << 1))/99/50.0
\end{aligned}
\tag{4.12}
$$

The numerator of the return value can therefore be calculated with a 6-input parallel adder (this IP block is configured to subtract the most significant input word, in this case $ipos^2 << 1$).

Ordinarily, we would want to combine the last two divisions into a division by 4950. However, this code actually performs an integer division, followed by a floating point division. Transforming this operation into a single floating point division results in slightly different answers. However, chaining an integer divider and a floating point divider together would result in extremely high latency. Luckily, as shown in [49], it is possible to perform integer division by multiplication (which will only require 1 cycle). So, the value of the numerator is multiplied by the constant[8] 0xA57EB503, then extracting the top 32 bits of the result and right-shifting by 6 places. This value is then

[8]Obtained using the tool at http://www.hackersdelight.org/magic.htm.

converted to floating point and divided by 50.0.

In the case that `dp` $\neq$ 2, the value of `i` in Listing 4.32 is calculated with a priority encoder. The `param` array is implemented as a lookup table, as is the linear interpolation in the else if statement (by precalculating the interpolation results). The call to `kf_erfc` is optimized by replacing `-(mean_diff-pshift)` with `(pshift-mean_diff)`. Figure 4.14 shows how both cases of the return value are calculated in hardware.

Listing 4.32: Calculation of the return value in the `dp` $\neq$ 2 case.

```
if ( dp>=200 ) i = nparam;
else
{
  for (i=0; i<nparam; i++)
    if ( param[i][0]>=dp ) break;
}
float pshift, pscale;
if ( i==nparam )
{
  pscale = param[nparam-1][1];
  pshift = param[nparam-1][2];
}
else if ( i>0 && param[i][0]!=dp )
{
  pscale = (param[i-1][1] + param[i][1])*0.5;
  pshift = (param[i-1][2] + param[i][2])*0.5;
}
else
{
  pscale = param[i][1];
  pshift = param[i][2];
}
return 0.5*kf_erfc(-(mean_diff-pshift)*pscale);
```

As for `kf_erfc` itself, this function does not offer many opportunities for hardware optimization due to its highly serial nature and reliance on double-precision floating point. So, the hardware simply sets a flag if `kf_erfc` should be calculated, and `kf_erfc` is called as a post-processing step. Leaving it in software does not negatively impact performance, as it was only responsible for approximately 0.003% of total run time.

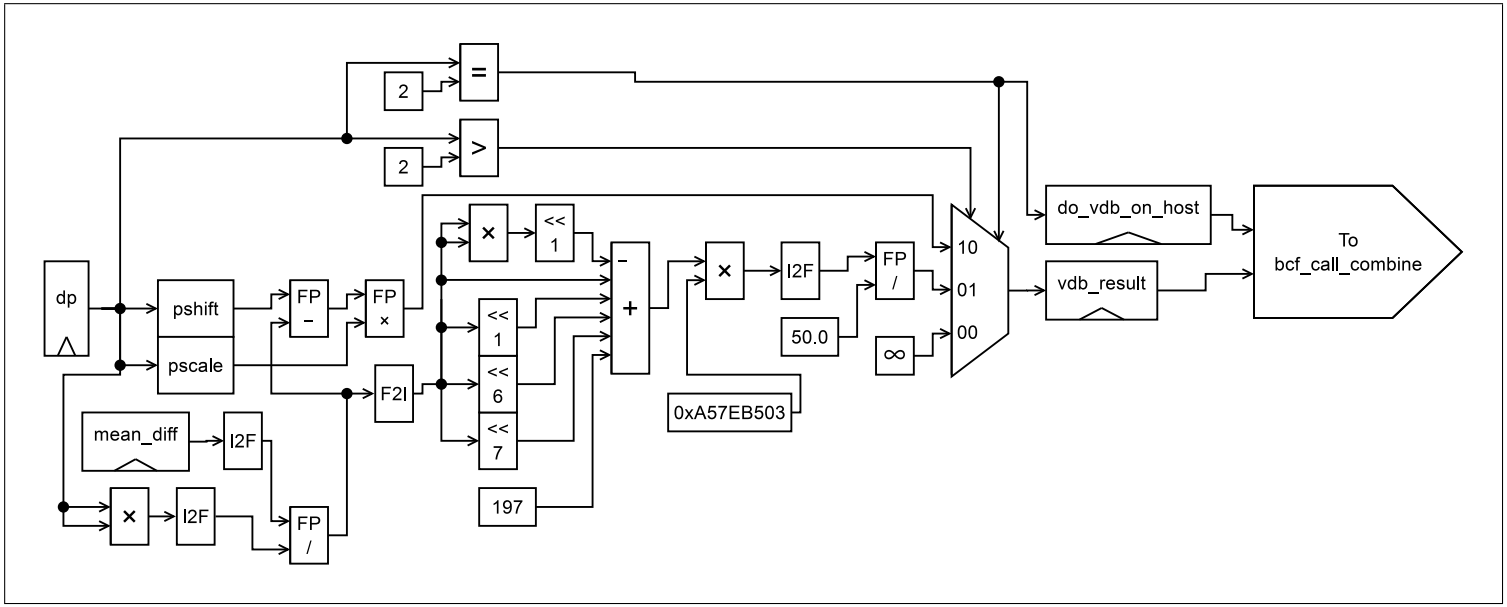

Figure 4.14: Block diagram of the hardware implementation of the final `calc_vdb` result calculation. "F2I" blocks convert from single-precision floats to integers, while "I2F" blocks convert from integers to single-precision floats. The 6-way parallel adder has a minus sign (-) on the most significant word to indicate that that word is actually subtracted.

CHAPTER 5

RESULTS AND DISCUSSION

5.1 Results

All performance tests were run on a workstation whose specifications are listed in Table 5.1. Every configuration tested was compiled with the default optimization level for `samtools`, -O2.

Table 5.1: Testbed Specifications

CPU	Intel Core i7-920
RAM	12GB DDR3-1066
HDD	2TB 7200RPM SATA 3.0 Gb/s
PCIe Capability	PCIe 2.0 x8
OS	Linux 3.16.5

5.1.1 Performance

Running `mpileup_call` against the full human genome HG00150 (the same as was used for profiling) required 7626.09 seconds, while the original SAMtools required 18428.76 seconds. The overall speedup was therefore 2.42x for `mpileup_call`. The hardware accelerated `mpileup_call_accel` required 6295.29 seconds, giving a speedup of 2.93x over the original SAMtools. These results are summarized in Figure 5.1 and Figure 5.2.

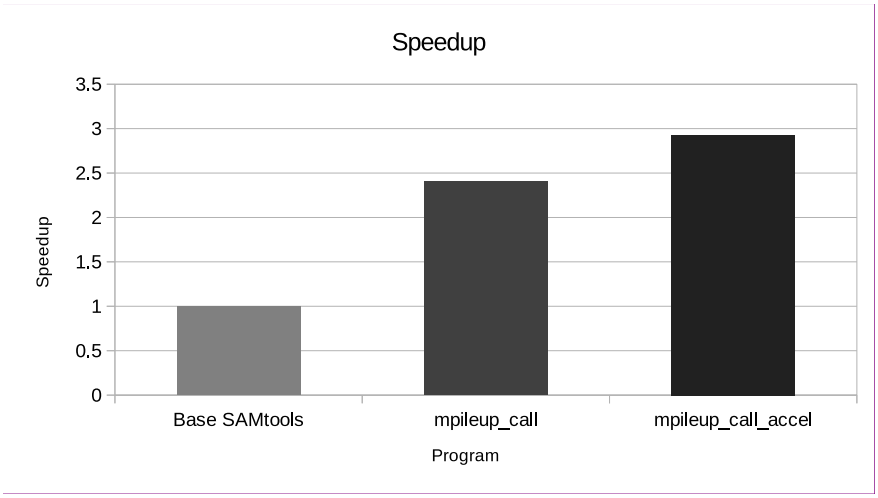

Figure 5.1: Speedup of `mpileup_call` and `mpileup_call_accel`.

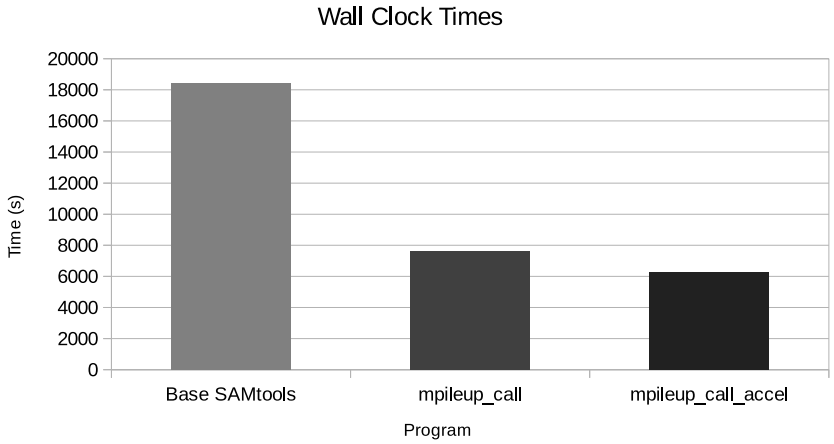

Figure 5.2: Wall Clock Time of `mpileup_call` and `mpileup_call_accel`.

5.1.2 Hardware Utilization

It was possible to fit three processing units on the Stratix V GX A7 FPGA. The overall resource utilization is shown in Table 5.2.

Table 5.2: FPGA Utilization

Resource	Utilization (%)
ALM	78
Memory	52
DSP Blocks	79

5.2 Discussion

5.2.1 Software Speedup

The speedup from combining `samtools mpileup` and `bcftools call` into a single tool was considerable, with a speedup of 2.42x over the base configuration. While much of this speedup comes from the elimination of the pipe between programs, it is not sufficient to explain the jump in performance. The other optimizations that were made, such as converting floating point variables containing integer values to integer types and simplifying `group_smpl` for the single sample case, must have had considerable effects as well.

5.2.2 Hardware Speedup

While the hardware accelerated version achieved a speedup of 2.93x over the base version of SAMtools, it is important to note that it only achieved a speedup of 1.21x over `mpileup_call`. The most likely reason for the low speedup is that communication between the host and accelerator becomes a bottleneck. First, the interface on the test machine is only capable of PCIe 2.0, while a machine that supported PCIe 3.0 would have twice the theoretical bandwidth. The lack of interrupt support from the DMA controller also hampers performance, since the host software must poll the accelerator for the done signal. The act of polling for the signal adds its own delay because the driver must read a memory location across the PCIe bus. The time

spent polling can be reduced somewhat by doing more work on the host side between sending data and checking for a response, but the amount of work done between transfers is limited by the sizes of the accelerator's FIFO buffers. The sizes of the FIFO buffers are in turn limited by resource and timing constraints. Transfers larger than the input FIFO's size must be broken up by the driver into multiple smaller transactions. Waiting for the FIFO to become ready between these transactions incurs additional delays for the same reason as checking the done signal.

In spite of these limitations, this design does indicate that FPGA acceleration of variant calling is a viable strategy. Further work could include modifying the DMA interface to enable interrupts and utilizing off-chip DRAM as an intermediate buffer for large transfers.

5.3 Applications

5.3.1 Genomic Cloud Computing

One potential application of this work would be in the development of large data centers for processing genomic data in the cloud. Utilizing hardware accelerators would allow a service provider to increase throughput without reducing the overall availability of compute nodes. FPGAs or ASICs utilizing this design would also use less power than GPU accelerators, reducing electricity costs and heat generation. However, the cost of FPGA accelerators is currently very high compared to the cost of a GPU or even an entire compute node, which could drastically limit how many a service provider would be willing to acquire. The limitation of only allowing single sample variant calling could be an issue in for this use case as well, since users may want to run large population-level variant calling workloads.

5.3.2 Personalized Medicine

Another possible application for this design is in personalized medicine. As the costs of sequencers decrease over time, the likelihood increases of them being acquired by hospitals or even individual doctors to better serve their patients. However, medical professionals would not necessarily have the technical expertise to run a variant calling workflow correctly. The costs of setting up and maintaining a compute infrastructure to provide adequate throughput would be another concern. One way to simplify the process would be to develop a "variant caller in a box" - a dedicated workstation containing hardware accelerators for each step of a variant calling workflow. Another option would be to integrate this hardware into the sequencer itself, allowing doctors to go from a sample of DNA to called variants in a single step. The cost of an FPGA accelerator is not as much of an issue in this scenario, as sequencers are currently an order of magnitude more expensive than FPGA accelerators.[1] Only allowing single sample workloads also matches up well for this use case, because personalized medicine practitioners would not be as concerned with running population-level studies.

[1] Current sequencers cost upwards of $80,000 [10], while the Terasic DE5 costs $8,000 [50].

CHAPTER 6

CONCLUSIONS

This project analyzed the performance of the SAMtools variant calling workflow in order to identify areas that could be parallelized with and offloaded to a hardware accelerator. It then proposed an alteration of the software architecture of SAMtools to reduce overhead and improve performance. It also presented a design for a hardware accelerator implemented on an Altera Stratix V GX A7 FPGA that parallelizes the most computationally intensive calculations in SAMtools.

The proposed software optimizations resulted in a speedup of 2.42x over the standard SAMtools workflow. The hardware accelerated version was able to achieve an overall speedup of 2.93x.

The speedup of this design is well below that of GPU-based variant callers (one achieved a speedup of 17.7x to 25.7x [28] and the other managed 42x to 50x [29]). However, both of these implementations targeted different algorithms that may not have been as well optimized as SAMtools, making it difficult to compare them fairly.

On the other hand, this work appears to be competitive with the Convey Computers HC-2 implementation of GATK, which achieved speedups of 3.7x (full genome) to 13x (single chromosome) [32]. The HC-2 utilizes 4 Xilinx Virtex-6 LX760 FPGAs, which have similar specifications to the Altera Stratix DE5 A7. Assuming that performance would scale linearly with the number of processing units, this work would potentially achieve a speedup of 11.71x for a full genome in a similar system.

This design also has potential to be extended in the future. Possible fu-

ture work includes implementing the design on several FPGAs to further increase performance. There are also forthcoming FPGA designs such as the Stratix 10 [51] that offer higher bandwidth, higher clock frequencies, and a massive increase in the amount of FPGA resources. These next generation FPGAs (or multiple current generation FPGAs) could provide sufficient area to implement significantly more processing elements, while increased on-chip memory would allow for larger transfer sizes and allow for the `beta` table to be implemented on-chip. This design could also be extended to take advantage of system-level innovations such as the Coherent Application Processor Interface (CAPI) [52] that has been introduced with IBM's POWER8 line of CPUs.[1]

[1]CAPI allows PCIe devices to coherently access system RAM.

REFERENCES

[1] M. L. Metzker, "Emerging technologies in DNA sequencing," *Genome Research*, vol. 15, no. 12, pp. 1767–1776, Dec 2005. [Online]. Available: http://dx.doi.org/10.1101/gr.3770505

[2] M. L. Metzker, "Sequencing technologies - the next generation," *Nature Reviews Genetics*, vol. 11, no. 1, pp. 31–46, Jan 2010. [Online]. Available: http://dx.doi.org/10.1038/nrg2626

[3] I. Toma, G. St Laurent, and T. A. McCaffrey, "Toward knowing the whole human: next-generation sequencing for personalized medicine," *Personalized Medicine*, vol. 8, no. 4, pp. 483–491, Jul 2011. [Online]. Available: http://dx.doi.org/10.2217/pme.11.27

[4] M. J. Puckelwartz, L. L. Pesce, V. Nelakuditi, L. Dellefave-Castillo, J. R. Golbus, S. M. Day, T. P. Cappola, G. W. Dorn, I. T. Foster, and E. M. McNally, "Supercomputing for the parallelization of whole genome analysis," *Bioinformatics*, vol. 30, no. 11, pp. 1508–1513, Jun 2014. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btu071

[5] "Variable Precision DSP Blocks in Stratix V Devices," 2014, Altera Corporation. [Online]. Available: http://www.altera.com/literature/hb/stratix-v/stx5_51004.pdf

[6] "7 Series DSP48E1 Slice," 2014, Xilinx, Inc. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

[7] R. Nielsen, J. S. Paul, A. Albrechtsen, and Y. S. Song, "Genotype and SNP calling from next-generation sequencing data," *Nature Reviews Genetics*, vol. 12, no. 6, pp. 443–451, Jun 2011. [Online]. Available: http://dx.doi.org/10.1038/nrg2986

[8] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences." *J Mol Biol*, vol. 147, no. 1, pp. 195–197, Mar 1981.

[9] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows-Wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, Jul 2009. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btp324

[10] M. Quail, M. E. Smith, P. Coupland, T. D. Otto, S. R. Harris, T. R. Connor, A. Bertoni, H. P. Swerdlow, and Y. Gu, "A tale of three next generation sequencing platforms: comparison of Ion torrent, pacific biosciences and illumina MiSeq sequencers," *BMC Genomics*, vol. 13, no. 1, p. 341, 2012. [Online]. Available: http://dx.doi.org/10.1186/1471-2164-13-341

[11] A. E. Minoche, J. C. Dohm, and H. Himmelbauer, "Evaluation of genomic high-throughput sequencing data generated on Illumina HiSeq and Genome Analyzer systems," *Genome Biology*, vol. 12, no. 11, p. R112, 2011. [Online]. Available: http://dx.doi.org/10.1186/gb-2011-12-11-r112

[12] S. Kosugi, S. Natsume, K. Yoshida, D. MacLean, L. Cano, S. Kamoun, and R. Terauchi, "Coval: Improving Alignment Quality and Variant Calling Accuracy for Next-Generation Sequencing Data," *PLoS ONE*, vol. 8, no. 10, p. e75402, Oct 2013. [Online]. Available: http://dx.doi.org/10.1371/journal.pone.0075402

[13] H. Li, "A statistical framework for SNP calling, mutation discovery, association mapping and population genetical parameter estimation from sequencing data," *Bioinformatics*, vol. 27, no. 21, pp. 2987–2993, Nov 2011. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btr509

[14] S. Pabinger, A. Dander, M. Fischer, R. Snajder, M. Sperk, M. Efremova, B. Krabichler, M. R. Speicher, J. Zschocke, and Z. Trajanoski, "A survey of tools for variant analysis of next-generation genome sequencing data," *Briefings in Bioinformatics*, vol. 15, no. 2, pp. 256–278, Mar 2014. [Online]. Available: http://dx.doi.org/10.1093/bib/bbs086

[15] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney, "High-throughput sequence alignment using Graphics Processing Units," *BMC Bioinformatics*, vol. 8, no. 1, p. 474, 2007. [Online]. Available: http://dx.doi.org/10.1186/1471-2105-8-474

[16] P. Klus, S. Lam, D. Lyberg, M. S. Cheung, G. Pullan, I. McFarlane, G. S. Yeo, and B. Y. Lam, "BarraCUDA - a fast short read sequence aligner using graphics processing units." *BMC Res Notes*, vol. 5, p. 27, 2012. [Online]. Available: http://dx.doi.org/10.1186/1756-0500-5-27

[17] C.-M. Liu, T. Wong, E. Wu, R. Luo, S.-M. Yiu, Y. Li, B. Wang, C. Yu, X. Chu, K. Zhao, R. Li, and T.-W. Lam, "SOAP3: ultra-fast GPU-based parallel alignment tool for short reads." *Bioinformatics*, vol. 28, no. 6, pp. 878–879, Mar 2012. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/bts061

[18] S. Kurtz, A. Phillippy, A. L. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. L. Salzberg, "Versatile and open software for comparing large genomes," *Genome Biology*, vol. 5, no. 2, p. R12, 2004. [Online]. Available: http://dx.doi.org/10.1186/gb-2004-5-2-r12

[19] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang, "SOAP2: an improved ultrafast tool for short read alignment," *Bioinformatics*, vol. 25, no. 15, pp. 1966–1967, Aug 2009. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btp336

[20] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome." *Genome Biol*, vol. 10, no. 3, p. R25, 2009. [Online]. Available: http://dx.doi.org/10.1186/gb-2009-10-3-r25

[21] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with Bowtie 2," *Nature Methods*, vol. 9, no. 4, pp. 357–359, Mar 2012. [Online]. Available: http://dx.doi.org/10.1038/nmeth.1923

[22] "Speeding Up the Next Generation Sequencing Pipeline," 2012, Convey Computer Corporation. [Online]. Available: http://www.conveycomputer.com/files/5313/8059/0389/CONV-13-048_DKFZ.pdf

[23] C. B. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, and W. L. Ruzzo, "Hardware Acceleration of Short Read Mapping," *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, 2012. [Online]. Available: http://dx.doi.org/10.1109/FCCM.2012.36

[24] "Accelerated BLAST Performance with Tera-BLAST™: a comparison of FPGA versus GPU and CPU BLAST implementations," May 2013, TimeLogic Division, Active Motif Inc. [Online]. Available: http://www.timelogic.com/documents/TimeLogic_Tera-BLAST_whitepaper_v1.0.pdf

[25] N. Homer, B. Merriman, and S. F. Nelson, "BFAST: An Alignment Tool for Large Scale Genome Resequencing," *PLoS ONE*, vol. 4, no. 11, p. e7767, Nov 2009. [Online]. Available: http://dx.doi.org/10.1371/journal.pone.0007767

[26] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool." *J Mol Biol*, vol. 215, no. 3, pp. 403–410, Oct 1990. [Online]. Available: http://dx.doi.org/10.1016/S0022-2836(05)80360-2

[27] "Hardware Acceleration," TimeLogic Division, Active Motif Inc., http://www.timelogic.com/catalog/775/hardware-acceleration.

[28] R. Jiang, F. Zeng, W. Zhang, X. Wu, and Z. Yu, "Accelerating Genome-Wide Association Studies Using CUDA Compatible Graphics Processing Units," *2009 International Joint Conference on Bioinformatics, Systems Biology and Intelligent Computing*, 2009. [Online]. Available: http://dx.doi.org/10.1109/IJCBS.2009.32

[29] M. Lu, J. Zhao, Q. Luo, B. Wang, S. Fu, and Z. Lin, "GSNP: A DNA Single-Nucleotide Polymorphism Detection System with GPU Acceleration," *2011 International Conference on Parallel Processing*, 2011. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2011.51

[30] R. Li, Y. Li, X. Fang, H. Yang, J. Wang, K. Kristiansen, and J. Wang, "SNP detection for massively parallel whole-genome resequencing," *Genome Research*, vol. 19, no. 6, pp. 1124–1132, Jun 2009. [Online]. Available: http://dx.doi.org/10.1101/gr.088013.108

[31] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, and M. A. DePristo, "The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data." *Genome Res*, vol. 20, no. 9, pp. 1297–1303, Sep 2010. [Online]. Available: http://dx.doi.org/10.1101/gr.107524.110

[32] M. Carneiro, "Accelerating variant calling," December 2013, Broad Institute. [Online]. Available: https://hpc.mssm.edu/files/Carneiro_workshop.pdf

[33] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin, "The Sequence Alignment/Map format and SAMtools," *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, Aug 2009. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btp352

[34] "Sequence Alignment/Map Format Specifcation," Sep 2014, The SAM/BAM Format Specification Working Group. [Online]. Available: https://samtools.github.io/hts-specs/SAMv1.pdf

[35] W. R. Pearson and D. J. Lipman, "Improved tools for biological sequence comparison." *Proc Natl Acad Sci U S A*, vol. 85, no. 8, pp. 2444–2448, Apr 1988.

[36] G. A. McVean, D. M. Altshuler (Co-Chair), R. M. Durbin (Co-Chair), G. R. Abecasis, D. R. Bentley, A. Chakravarti, A. G. Clark, P. Donnelly, E. E. Eichler, P. Flicek, and et al., "An integrated map of genetic variation from 1,092 human genomes," *Nature*, vol. 491, no. 7422, pp. 56–65, Oct 2012. [Online]. Available: http://dx.doi.org/10.1038/nature11632

[37] B. Ewing and P. Green, "Base-calling of automated sequencer traces using phred. II. Error probabilities." *Genome Res*, vol. 8, no. 3, pp. 186–194, Mar 1998.

[38] R. Durbin, "Segregation based metric for variant call QC," June 2014, SAMtools. [Online]. Available: https://samtools.github.io/bcftools/rd-SegBias.pdf

[39] H. B. Mann and D. R. Whitney, "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, Mar 1947. [Online]. Available: http://dx.doi.org/10.1214/aoms/1177730491

[40] P. Danecek, C. Nellåker, R. E. McIntyre, J. E. Buendia-Buendia, S. Bumpstead, C. P. Ponting, J. Flint, R. Durbin, T. M. Keane, and D. J. Adams, "High levels of RNA-editing site conservation amongst 15 laboratory mouse strains." *Genome Biol*, vol. 13, no. 4, p. 26, 2012. [Online]. Available: http://dx.doi.org/10.1186/gb-2012-13-4-r26

[41] E. A. M. Banks, "BCF (Binary VCF) version 2," October 2013, 1000 Genomes. [Online]. Available: http://www.1000genomes.org/wiki/analysis/variant-call-format/bcf-binary-vcf-version-2

[42] P. Danecek, A. Auton, G. Abecasis, C. A. Albers, E. Banks, M. A. DePristo, R. E. Handsaker, G. Lunter, G. T. Marth, S. T. Sherry, and et al., "The variant call format and VCFtools," *Bioinformatics*, vol. 27, no. 15, pp. 2156–2158, Aug 2011. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btr330

[43] "Multisample SNP Calling," 12 2010, SAMtools. [Online]. Available: http://samtools.sourceforge.net/mpileup.shtml

[44] H. Li, "Improving SNP discovery by base alignment quality," *Bioinformatics*, vol. 27, no. 8, pp. 1157–1158, Apr 2011. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btr076

[45] R. P. Brent, *Algorithms for Minimisation without Derivatives (Automatic Computation)*. Prentice Hall, 1972.

[46] G. H. Hardy, "Mendelian Proportions In a Mixed Population," *Science*, vol. 28, no. 706, pp. 49–50, Jul 1908. [Online]. Available: http://dx.doi.org/10.1126/science.28.706.49

[47] L. Mainzer and G. Rendon, private communication, April 2014.

[48] "AN 690: PCI Express DMA Reference Design for Stratix V Devices," July 2014, Altera. [Online]. Available: http://www.altera.com/literature/an/an690.pdf

[49] H. Warren, *Hacker's Delight*. Upper Saddle River, NJ: Addison-Wesley, 2013.

[50] "DE5-Net FPGA Development Kit," Terasic Technologies, Inc. [Online]. Available: http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=526&PartNo=7

[51] "Stratix 10 FPGAs and SoCs: Delivering the Unimaginable," 2014, Altera Corporation. [Online]. Available: http://www.altera.com/devices/fpga/stratix-fpgas/stratix10/stx10-index.jsp

[52] J. Stuecheli, "Power Technology For a Smarter Future," IBM Corporation. [Online]. Available: https://www.ibm.com/developerworks/community/wikis/form/anonymous/api/wiki/61ad9cf2-c6a3-4d2c-b779-61ff0266d32a/page/1cb956e8-4160-4bea-a956-e51490c2b920/attachment/16bd2505-d09b-4ec6-bdbc-8ca1c668e602/media/POWER8_VUG.pdf